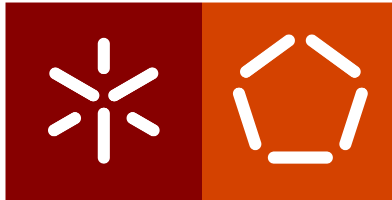
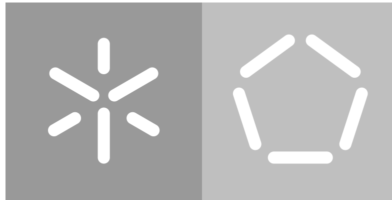


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Gonçalo Rodrigues Pinto

Programming language complexity analysis and its impact on Checkmarx activities

October 2022



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Gonçalo Rodrigues Pinto

Programming language complexity analysis and its impact on Checkmarx activities

Master dissertation
Integrated Master's in Informatics Engineering

Supervisors:
Pedro Rangel Henriques, UM
Daniela da Cruz & João Cruz, Checkmarx

October 2022

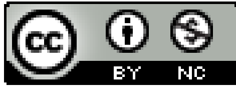
AUTHOR COPYRIGHTS AND TERMS OF USAGE BY THIRD PARTIES

This is an academic work which can be utilised by third parties given that the rules and good practices internationally accepted, regarding author copyrights and related copyrights.

Therefore, the present work can be utilised according to the terms provided in the license bellow.

If the user needs permission to use the work in conditions not foreseen by the licensing indicated, the user should contact the author, through the RepositóriUM of University of Minho.

License provided to the users of this work



Attribution-NonCommercial
CC BY-NC

https://creativecommons.org/licenses/by-nc/4.0/

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Gonçalo Rodrigues Pinto

ABSTRACT

Tools for Programming Languages processing, like Static Analysers (for instance, a Static Application Security Testing (SAST) tool, one of Checkmarx's main products), must be adapted to cope with a given input when the source programming language changes.

Complexity of the programming language is one of the key factors that deeply impact the time of giving support to it.

This Master's Project aims at proposing an approach for assessing language complexity, measuring, at a first stage, the complexity of its underlying context-free grammar (CFG).

From the analysis of concrete case studies, factors have been identified that make the support process more time-consuming, in particular in the stages of language recognition and in the transformation to an abstract syntax tree (AST). In this sense, at a second stage, a set of language features is analysed in order to take into account the referred factors that also impact on the language processing.

The main objective of the Master's work here reported is to help development teams to improve the estimation of time and effort needed to adapt the SAST Tool in order to cope with a new programming language.

In this dissertation a tool is proposed, that allows for the evaluation of the complexity of a language based on a set of metrics to classify the complexity of its grammar, along with a set of language properties. The tool compares the new language complexity so far determined with previously supported languages, to predict the effort to process the new language.

Keywords: Complexity, Grammar, Language-based-Tool, Programming Language, Static code analysis.

RESUMO

Ferramentas para processamento de Linguagens de Programação, como os Analisadores Estáticos (por exemplo, uma ferramenta de Testes Estáticos para Análise da Segurança de Aplicações, um dos principais produtos da Checkmarx), devem ser adaptadas para lidar com uma dada entrada quando a linguagem de programação de origem muda.

A complexidade da linguagem de programação é um dos fatores-chave que influencia profundamente o tempo de suporte à mesma.

Este projeto de Mestrado visa propor uma abordagem para avaliar a complexidade de uma linguagem de programação, medindo, numa primeira fase, a complexidade da gramática independente de contexto (GIC) subjacente.

A partir da análise de casos concretos, foram identificados fatores (relacionados como facilidades específicas oferecidas pela linguagem) que tornam o processo de suporte mais demorado, em particular nas fases de reconhecimento da linguagem e na transformação para uma árvore de sintaxe abstrata (AST). Neste sentido, numa segunda fase, foi identificado um conjunto de características linguísticas de modo a ter em conta os referidos fatores que também têm impacto no processamento da linguagem.

O principal objetivo do trabalho de mestrado aqui relatado é auxiliar as equipas de desenvolvimento a melhorar a estimativa do tempo e esforço necessários para adaptar a ferramenta SAST de modo a lidar com uma nova linguagem de programação.

Como resultado deste projeto, tal como se descreve na dissertação, é proposta uma ferramenta, que permite a avaliação da complexidade de uma linguagem com base num conjunto de métricas para classificar a complexidade da sua gramática, e em um conjunto de propriedades linguísticas. A ferramenta compara a complexidade da nova linguagem, avaliada por aplicação do processo referido, com as linguagens anteriormente suportadas, para prever o esforço para processar a nova linguagem.

Palavras-Chave: Complexidade, Gramática, Ferramenta baseada em Linguagens, Linguagem de Programação, Análise de código estático.

CONTENTS

1 Introduction 1
1.1 Context 1
1.2 Motivation 2
1.3 Objectives 2
1.4 Research Hypothesis 3
1.5 Document Structure 3
2 State of the Art 5
2.1 Software Complexity 5
2.1.1 Measure Software 6
2.1.2 Importance of Measurement 7
2.1.3 A Tool for Software Measurement 7
2.1.4 Measures of Software Complexity 8
2.2 Grammar Complexity 8
2.2.1 Grammar Definitions 9
2.2.2 Measuring Grammar Complexity 14
2.3 Language Complexity 16
2.3.1 Checkmarx CxSAST 16
2.3.2 Supporting of Programming Languages by CxSAST 19
3 Proposal 26
3.1 A DSL to describe the properties of a Programming Language 27
3.1.1 Parser Rules 28
3.1.2 Example 30
4 Language Complexity Evaluator 31
4.1 Architecture and Technologies 31
4.2 Backend 32
4.2.1 Data Models 33
4.2.2 DAOs 35
4.2.3 ANTLR 36
4.2.4 Services 37
4.2.5 Controllers 37
4.3 Frontend 39
4.4 Predict effort to adapt CxSAST tool 44
5 Case Study 45

5.1 Programming Language Analysis 45
6 Conclusion 53
6.1 Future Work 55

A ANother Tool for Language Recognition (ANTLR) 59
A.1 Components 60
A.2 Grammar 61
A.2.1 Lexicon 61
A.3 Structure 62
A.4 Parser Rules 62
A.5 Lexer Rules 62
B PropLD Lexical Rules 63

LIST OF FIGURES

Figure 1	An abstract, high-level model of a SAST tool.	17
Figure 2	An abstract, high-level model of the SAST tool builder.	18
Figure 3	Language Complexity Evaluator, architecture.	26
Figure 4	PropLD Syntax Diagram.	27
Figure 5	Diagram showing LCE architecture and technologies used.	31
Figure 6	Backend structural diagram.	33
Figure 7	Diagram of the Data Models component's structure.	34
Figure 8	Main page.	40
Figure 9	Help page.	40
Figure 10	About page.	41
Figure 11	LCE operations available in format of a carousel.	42
Figure 12	Operations available in format of a sidebar list.	42
Figure 13	Visual representation of the grammar.	43
Figure 14	Visual representation of the DGS.	43
Figure 15	Grammar size metrics for Python evaluated by LCE tool.	46
Figure 16	Grammar structure for Python evaluated by LCE tool.	47
Figure 17	Information extracted from Python features by LCE tool.	48
Figure 18	Final Considerations about Python drawn by LCE tool.	50
Figure 19	Comparison of results between Python and JavaScript.	52
Figure 20	ANTLR creates a variety of files from the grammar.	60

LIST OF TABLES

Table 1 Metrics for evaluating the Size of Context-Free Grammars. 14
Table 2 Metrics for evaluating the Structure of Context-Free Grammars. 15

ACRONYMS

A

ANTLR ANother Tool for Language Recognition.

AST Abstract Syntax Tree.

C

CFG Context-Free Grammar.

D

DAST Dynamic Application Security Testing.

DGS Dependency Graph between Symbols.

DOM Document Object Model.

I

IEC International Electrotechnical Commission.

ISO International Organization for Standardization.

S

SAST Static Application Security Testing.

SDLC Software Development Life Cycle.

1

INTRODUCTION

Engineering has been around since ancient times. The term "software" was first used in a computer context by American statistician John W. Tukey in his book "The Teaching of Concrete Mathematics" (Tukey, 1958).

However, it was in the so-called software crisis[1] of the 60's that the term '*software engineering*' was coined, identifying many of the problems in software development. That term is related to some issues with software going over budget and schedule, not meeting customer requirements, and requiring large amounts of maintenance (where it was possible). Software engineering is an effort to address these issues and aims to improve the quality of software products (Schach, 2010).

One of the most crucial phases in software engineering is maintenance. ISO and IEC define software maintenance as the change of a software product after distribution, to correct flaws or increase performance (ISO/IEC, 2006).

Several new practices have emerged in recent years that can improve software maintenance. The major consideration is how to balance the enormous complexity of software with the cost, effort, and time required for maintenance.

1.1 CONTEXT

A Static Application Security Testing (SAST) tool analyses source code written in a programming language and finds its security vulnerabilities. While this solution satisfies the need (detecting software vulnerabilities), there are other factors that need special attention in this type of tool, one of which is the (evolutive) maintenance required.

For that, it must be adapted to handle different inputs whenever the source programming language changes. To do this, one of the first steps towards supporting a new programming language in this tool is to create a new parser to analyse the respective source code.

The complexity of the programming language is one of the key factors that affects the time to provide support for it. This fact raises the need to evaluate whether the complexity of a programming language is related to the complexity of its context-free grammar.

1 For more information and other details, consult the link https://www.hackreactor.com/blog/the-history-of-coding-and-software-engineering

1.2 MOTIVATION

During the life cycle of a static analysis application, many modifications, and extensions are required to support increasing business requirements. One of the most difficult requirements of supporting a programming language in a static analysis tool is to predict its cost in terms of time and effort required, due to the many factors and components that need to be considered.

Today's programming languages are the product of developments that began in the 50's. Conservative estimates show that there are approximately 700 programming languages[2]. New concepts, languages, and paradigms are still being developed.

In a sector where the requirements and needs of users are changing, it is essential to be aware of the new programming languages that emerge and their advantages over others, as they can bring benefits and be better suited to the new requirements compared to the languages already in use.

The potential benefits include an increased ability to express ideas or a smother learning curve, or, better experience in choosing appropriate languages leading to a general advancement in computing, such as in the area of *quantum computation* that is emerging.

Customers are also aware of these advances, so they develop code on these latest platforms with new languages, expecting it to be functional, reliable, efficient, and most importantly, secure. To achieve this objective, they rely on companies that provide SAST tools, such as Checkmarx, to detect as many vulnerabilities as possible.

It is then up to the software developers to know the cost of supporting a new language in their tool. Forecasting effort and time needed for the adaptation is crucial for both customers and development teams because it allows for better planning, task control, prioritization. Actually, a correct prediction supports a better inform decision process. Therefore, this type of software continues to develop through maintenance after its initial development.

Thus, given the difficulties associated with the SAST engine in analysing and supporting a new programming language, it is motivating to create a tool that selects and implements a set of metrics and analyses a set of features to assess the complexity of a language, in this way, it is possible to understand the language in some aspects, such as the effort and time required to support it.

1.3 OBJECTIVES

The primary purpose of the Master's work described here is to assist language support teams in better estimating the time and effort required to support a new programming language in a static analyser.

---

2 To learn more, see https://careerkarma.com/blog/how-many-coding-languages-are-there/

This task includes five more specific objectives:

1. Identify the metrics that shall be used to measure the syntactic complexity of a language;
2. Implement an automatic evaluator to compute the metrics referred;
3. Identify other language features that impact on the processing of the language sentences;
4. Implement an analyser for those features;
5. Develop a repository to compare a new language with previously processed languages.

Along the dissertation, we propose and present a tool for evaluating the difficulty of supporting a language based on a collection of metrics to classify the complexity of its grammar, as well as a set of features. To forecast the work required to process the new language, the program compares the new language properties so far identified with previously supported languages.

1.4 RESEARCH HYPOTHESIS

With this Master's work, it is intended to prove that it is possible to evaluate language complexity, through the complexity of its underlying context-free grammar (CFG) and a set of linguistic features, which involves a set of factors that make the support process more time-consuming.

1.5 DOCUMENT STRUCTURE

This Chapter 1 discussed the significance of maintenance, what a SAST tool is and its limits, how complexity is measured, and the objectives of the tool to be the provided tool.

In Chapter 2, it is intended to focus on the main concepts of software, language, and grammar that help in determining the complexity of programming languages and their impact on processing.

After the introduction of the concepts, Chapter 3 focuses on explaining the strategy adopted, the architecture of the system to prove the thesis hypothesis. The DSL created to describe the language's lexical and semantical features is presented. This DSL actually allows for the definition of the extra-grammatical features that have to be described by those who know the language.

Introduced and described the language created for this particular problem domain, it is fundamental to present the tool developed, showing its architecture, technologies and

components to produce a quantitative and qualitative report of the language, this information is described in Chapter 4.

Following the presentation of the tool for assessing the complexity of a programming language and its influence on its support, Chapter 5 will discuss a case study involving a real language. The tool's main results will be provided.

Finally, Chapter 6 is the summary of the dissertation. Some conclusions and results achieved are emphasized; and a description of future work is also included.

2

STATE OF THE ART

Chapter 2 begins by introducing the concept of software complexity and its impact on the timing of support. After that, one of the tools that allows to evaluate the complexity of a language and grammars, is presented, explaining its relation with languages and how grammatical complexity is defined. Afterwards, the way to measure this grammatical complexity, by metrics, is presented. Finally, the subject of this project, complexity of programming languages, is introduced.

2.1 SOFTWARE COMPLEXITY

In recent years, software complexity has been the subject of much interest in order to define metrics for measuring it. Complexity is the characteristic associated with a system or model whose state is composed of many parts and is difficult to understand or find an answer for.

Understanding and measuring the software complexity is not something simple and obvious. Nevertheless, it is important to keep in mind that complexity is in permanent evolution throughout the time of the project.

That said, complexity can be created at different phases of the software life cycle (phase of requirements, development, testing and integration, and of course the maintenance phase). The authors Nystedt and Sandros (1999) suggest a two-pronged classification of complexity:

- The intrinsic **complexity of the problem**, which was developed during the requirements phase;
- After this phase, the **complexity of the solution** is increased during the development stages and in the remaining phases.

As previously said, software complexity is difficult to quantify, but considering its importance, Fenton and Pfleeger (1997) suggests that it may be divided into four types:

- **Problem complexity**, measures the complexity of the underlying problem;
- **Algorithmic complexity**, measures the complexity of the algorithm implemented to solve the problem;

- **Structural complexity**, measures the structure of the software used to implement the algorithm;
- **Cognitive complexity**, measures the effort required to understand the software.

On one hand, the complexity of the problem is related to the complexity that is generated by the requirements. On the other hand, the complexity of a solution can be measured through algorithmic, structural, and cognitive complexity. Therefore, during the development process, the types of complexity that can be measurable are only the algorithmic and structural ones.

However, measuring the complexity of the software is useful, as it may predict the effort needed for a project. By comparing the problems and considering the solutions found so far, it is possible to predict the properties of the solution a new problem, such as cost or time.

2.1.1 *Measure Software*

Software measurement has become essential to software engineering (Coleman et al., 1994; Finkbine, 1996; Fenton and Pfleeger, 1997; Fenton and Neil, 2000; Kan, 2002; Chopra and Sachdeva, 2015).

In order to ensure that the previously stated goal[1] of software engineering is achieved, the use of measurement is essential. Here are some examples of their importance by the entities involved at the time of support:

- In order to verify that the requirements are consistent and complete, development teams measure characteristics of the software they have produced;
- In the sense of realizing that the software is ready and does not exceed the stipulated budget, the project managers measure the attributes of their software;
- Customers use measurement to determine whether the product they have bought/ordered meets the requested requirements and is of sufficient quality.
- Even after the software is released, it is necessary to follow up by making continuous evaluations in order to see where the product can be improved or updated;

Knowledge about the properties of entities is obtained through measurement. In order to relate and compare properties between entities, rules are used. Nevertheless, is not something clear or easy to define, because it is always open to subjective interpretation.

Every time we effectively measure something that was not measurable at first glance, we expand the power of software engineering, as is done in other disciplines in this area.

1 Produce flawless software, followed by a spending plan and on time, that meets the customer's requirements.

Despite the opinion of some authors that an attribute such as maintainability is simply not quantifiable, the basic truth of proposing a set of metrics allows us to use measurement to build our understanding of it.

2.1.2 *Importance of Measurement*

With a more stabilized idea of what measurement is, it is necessary to understand the reason for using measurement in software because science and engineering can be neither practical nor effective without it.

Although its use is not mandatory, it is not always known what complicates a project for this reason measurement helps at least to assess the state of the product, process, and resources of the projects, thus allowing short-term and long-term control. To prove their importance in software engineering, Fenton and Pfleeger (1997) list three basic activities for which measurements are important:

1. **Understanding** by evaluating the product at its current stage of development and maintenance helps you establish guidelines for setting expected future goals.
2. Based on this understanding of the system, it is possible to **control** what is happening in the software, foreseeing possible harmful events to the system and changing to prevent them from happening, thus helping to achieve the defined objectives.
3. Finally, by combining the understanding with the control of the software, it is possible to **improve** previously identified software weaknesses, failures and shortcomings, thus making the system better quality and more efficient.

2.1.3 *A Tool for Software Measurement*

As introduced earlier, in software engineering, most qualities are not directly quantifiable. There is no theory that shows whether a set of metrics is valid. We only know that there is a structure based on objectives for software measurement, which can improve software engineering practices.

This structure is based on three principles: categorizing the entities to be investigated, determining relevant measuring targets, and determining the maturity level attained.

In software engineering problems, the crucial question arises in determining which attributes should be measured and how these can be applied.

2.1.4 *Measures of Software Complexity*

Size along with structure are the main internal properties in measuring software complexity, according to Fenton and Pfleeger (1997).

- **Size Complexity**, the traditional attribute to measure in software, because it is advantageous, accessible to measure without having to run the system, and because software development is a physical entity:
  - The physical size is defined by length. The code (including traditional measures such as lines of code (LOC), comment lines, data declarations, and others), the specification, and the design (the specification length can be a good indicator of how long the design is likely to be, and the same can be said about the design length relative to code length) are the three aspects whose size is worth knowing.
  - What is truly extracted by the user is functionality. Three ways are used to quantify functionality: function points, which link the functionality to the specification; object points; and specification weight.
- **Structural Complexity**, determines the level of project productivity, as it has been proven that a larger module does not always take longer to specify, design, code, and test than a small one. The structure affects its maintenance and development effort:
  - Control-Flow structure illustrates the program's interactive and looping nature, which is generally assessed using direct graphs;
  - Data-Flow structure tracks the behaviour of data as it interacts with the program;
  - Data Structure, independent of software, assesses the arrangement of the data itself.

Therefore, complexity can be assessed by quantifying a subset of software metrics that are based on static analysis. In this way, we can better understand the language in some aspects, such as the size and structure.

2.2 GRAMMAR COMPLEXITY

Grammars formally specify languages, so the complexity of languages depends on the complexity of grammars, even if the complexity of grammars does not fully imply the complexity of language.

In this context, the use of grammars is proposed to define the languages and support their recognition, which leads to a strong relationship between grammar and the language that is defined by that grammar (Henriques, 2013). Therefore, grammar will be one tool to assess the complexity of a language.

2.2.1 *Grammar Definitions*

Before moving into the subject of complexity, the focus of this Subsection is to present the basic definitions of grammar. It is impossible to approach this subject without mentioning the author who made the automatic processing of artificial languages possible, the American philosopher-analyst Avram Noam Chomsky, considered by many to be "the father of modern linguistics".

Noam Chomsky handled the creation of the concept of generative transformational grammars, organizing grammars into 4 large classes in the famous so-called Chomsky Hierarchy[2]. This Chomsky grammatical model, known as CFG (Chomsky, 1956; Backhouse, 1979; Knuth, 2005), usually in software systems, defines the syntax of the programming language.

Definition 1 (CFG). *A Context-Free Grammar (CFG) is defined by the four-tuple:*

$$CFG =< T, N, S, P >$$

where,

- ***T** is the set of **terminal symbol**s of the language (the alphabet or vocabulary).*

  *The set **T** of terminal symbols is divided into 3 disjoint subsets -* $T = RW \cup Sig \cup TV$ *- of **Reserved-Words, Signs, and Terminal-Variables**;*

- ***N** is the set of **non-terminal** symbols of the grammar;*

- $\boldsymbol{S} \in \boldsymbol{N}$ *is the **initial symbol** or axiom of the grammar;*

- ***P** is the **set of productions** or derivation rules of grammar.*

  Every production $\boldsymbol{p} \in \boldsymbol{P}$ *is a rule of the form:*

  $$p : X_0 \to X_1 ... X_i ... X_n$$

  where p is the rule identifier, $\to$ *is the derivation operator,* $X_0 \in N$ *e* $X_i \in (N \cup T), 1 \leq i \leq n$

  *In a production **p** must be interpreted from the left side of the operator, also called LHS(p) which is always a non-terminal symbol, on the right side of the operator RHS(p) is a sequence of non-terminal and terminal symbols.*

After defining a production rule, it is now relevant to characterize a production unit, as this will be crucial in the following definitions.

2 According to Chomsky Hierarchy, grammar is divided into 4 levels: Type-0 Unrestricted grammars, includes all formal grammars and there are no restrictions on the grammatical rules of this type of language; Type-1 Context-sensitive grammars are used to represent context-sensitive languages; Type-2 Context-free grammars, which generate Context-free languages, recognized by a non-deterministic push-down automaton; and Type-3 Regular grammars, which generate regular languages, decided by a finite state automaton.

Definition 2 (Unit production)**.** *Unit production is a production up∈P with a single symbol to the right ($\#RHS(p) = 1$) of the form:*

$$X_0 \to X_1$$

which is the only alternative for X_0 (note that it will always be $X_0 \neq S$). The set of these productions will be denoted by UP, where $UP \in P$.

The following is a small example of a CFG, serving as a simple case study, that defines a language called Mixed List Language.

Example 1. *The grammar listed below formally describes a language for writing lists of elements (one or more) written between the reserved words START and END, where each element of the list can be either a word(wrd) or an integer(num).*

```
T = { num, wrd, ',', 'START', 'END' }
N = { Elem, Content, List }
S = List
P = {
      p0: List --> 'START' Content 'END'
      p1: Content --> Elem
      p2: Content --> Elem ',' Content
      p4: Elem --> wrd
      p5: Elem --> num
    }
```

According to this example, a list must always contain elements of any type and in any order. These are valid sentences from the language generated by this grammar:

```
START 1 END
START example END
START is, 1, example, with, 6, words END
```

and an invalid sentence is, for example:

```
END invalid START
START invalid sentence END
START , invalid, sentence END
```

Since a grammar is a specification, it applies to have a way to validate that it is correct, since it is impossible to prove that semantically this specification conveys the idea of its creator. The notion of a well-formed grammar is presented.

Definition 3 (CFG well-formed)**.** *A Context-Free Grammar is said to be well-formed if:*

- *for all $X \in N$, there is at least 1 production with X on the left;*
- *for all $X \in N$, X is **reachable**, i.e., there is at least 1 derivation from the axiom that uses X;*
- *for all $X \in N$, X is **terminable** according to the definition below.*

Complementing the previous definition,

Definition 4 (Terminable Symbol)**.** *A grammatical symbol is said to be **terminable** if:*

- *is a terminal symbol.*
- *is a non-terminal symbol and there is at least 1 production with that symbol on the left is a terminable sequence.*

*where a **sequence of symbols $N \cup T$ is terminable** if*

- *the sequence is empty;*
- *each symbol in this sequence is terminable.*

To clarify the terms introduced before, an example is presented below.

Example 2. *Despite being unhelpful, the List language grammar, shown above in Example 1, is well-formed because:*

- *there is at least one production for each one of the 3 non-terminals symbols:*

  `p0 for List; p1 and p2 to Content; p3 and p4 for Elem;`

- *the two non-terminals beyond the axiom are reachable from List:*

  `Content is used directly on p0; Elem is used indirectly (via Content) on p1 or p2.`

- all non-terminals are terminable:

  ```
  Elem is terminable thanks to p3 or p4 because his RHS is a terminal;
  Content is terminable because p1 in which the RHS is a terminable sequence;
  List is terminable because in the RHS(p0) all symbols are terminables.
  ```

To aid comprehension, further on in the characterization of grammars, the concept of a dependency graph between symbols is introduced.

Definition 5 (DGS)**.** *A graph is called a **Dependency Graph between Symbols** (DGS) if its vertices are the symbols $N \cup T$ of the grammar and the branches or arcs, go from Y vertex to X vertex whenever there exists in P a production p of the form $X \rightarrow ...Y...$, and it is then said that X **depends on** Y (because Y enters the definition of X) or that Y **derived from** X.*

Another fundamental concept associated with grammars, both from the theoretical point of view of formalization and from the more practical perspective of processor implementation, is the notion of derivation tree, which is defined below.

Definition 6 (Derivation Tree of a production)**.** *Associated with each production $p \in P$ of the grammar is a* ***derivation tree*** *whose root is X_0 and the descendants, or children, are the n symbols X_i on the right-hand side.*

The following example clarifies this definition.

Example 3. *In the case of the List grammar of the Example 1, the DGS consists of the following branches:*

```
('START', List)
(Content, List)
('END', List)
(Elem, Content)
(',', Content)
(Content, Content)
(wrd, Elem)
(num, Elem)
```

The derivation trees of productions can be joined by pasting the leaf X_i of one tree, while another tree that has X_i as its root. In this way, a complete **derivation tree** (DA) is built.

Definition 7 (Derivation Tree)**.** *Starting with a tree whose root is the initial symbol of the grammar, a tree corresponding to a production such as the axiom on the left, one replaces each of its non-terminal descendants with a tree having that symbol as its root, and repeats the process successively and recursively until all leaves are terminal symbols. The complete tree thus formed from the axiom is a grammar* ***derivation tree****.*

From this arises the notion of a valid sentence.

Definition 8 (Valid Sentence)**.** *The border of a derivation tree, the sequence of terminal symbols got by concatenation of the leaves, collected from left to right, is a* ***valid sentence of the language****. A sentence (sequence of terminals) is only valid, only belongs to the language, if it is the frontier of a derivation tree of this grammar.*

Thus, generating (or deriving) a sentence corresponds to building a derivation tree; and validating a sentence (accepting it as belonging to a language) corresponds to discovering a derivation tree for that sentence.

Definition 9 (Non-ambiguous Language). *A language is said to be* ***non-ambiguous*** *if, for every valid sentence, there is one and only one derivation tree.*

To calculate the structural metrics for grammars, a few more notions must be introduced. One is the grammatical level idea suggested by Csuhaj-Varjú and Kelemenová (1993), and the other is another representation in the form of a graph termed a call graph.

Definition 10 (Immediate Successor). *It is said that B is A's immediate successor if non-terminal A derives some sequence of symbols β and β contains some non-terminal B. This is shown by the notation $A \rhd B$.*

Definition 11 (Successor). *It is said that C is A's successor if β derives some sequence of symbols γ and γ contains some non-terminal C. This is shown by the notation $A \rhd^* C$.*

Definition 12 (Grammatical Levels). *If $A \rhd^* C$ and $C \rhd^* A$, then the successor relation produces an equivalence relation on the non-terminals, and it is written $A \equiv C$, then A is equivalent to C. In the context of grammar non-terminals, every equivalent relation on a set divides that set into a number of equivalence classes described as grammatical levels.*

Definition 13 (Call Graph). *In a call graph, the edges show a successor relation between a non-terminal on the left side and then a non-terminal on the right side. The nodes in the graph are non-terminals.*

This concept is made clearer by the example below.

Example 4. *Given the definitions from 10 to 12, we can state that:*

- `Content is a immediate successor of List`
  `Elem is also a immediate sucessor of Content.`
- `Elem is a successor of List.`
- `Considering the succession relations present this grammar gives rise to`
  `3 different levels each of size 1.`

In the case of the List grammar of the Example 1, the call graph consists of the following branches:

```
(Content, List)
(Elem, Content)
(Content, Content)
```

2.2.2 *Measuring Grammar Complexity*

Since any grammar characterizes a language and gave a premise for determining elements of that language, a grammar might be considered as both a program and a specification. Considering what has been previously presented to show the relationship between grammars and languages, supporting a new programming language in a static analysis tool is faster and requires less effort, the less complex the grammar is.

The complexity of a grammar as a producer of a language that directs the recognition of sentences in that language concerns its size and how the symbols depend on each other, i.e., the number of symbols on the right-hand side of a production for a given symbol on the left-hand side, or how many symbols that symbol intervenes in.

Considering this, the need to evaluate the complexity of a grammar arises, since it allows us to evaluate the complexity of the language defined by it. Thus, the use of grammatical metrics is relevant to the study in question.

The metrics for evaluating the complexity of a well-formed context-free grammar are presented, dividing them into the previously mentioned criteria:

- **Size Metrics** that measure the number of symbols (terminals or non-terminals) and productions used to write the grammar. As the grammar is the basis to recognize the sentences of the language defined by itself, it is reasonable to state that the size of the grammar has a direct impact on the time and effort necessary to support that language.

Metric	Definition
#P	Number of productions
#N	Number of non-terminals
#T	Number of terminals
#UP	Number of unit productions
RHS-Max	Maximum number of symbols on an RHS
RHS[1]	Average number of symbols in the RHS
ALT[2]	For the same left sides, average size of alternative productions
MCC[3]	McCabe Complexity

Table 1: Metrics for evaluating the Size of Context-Free Grammars.

Notes:

1. To compute the average number of symbols in the RHS, we calculate the following equation

$$RHS = \frac{\sum_{(n \to \alpha) \in P} size(\alpha)}{\#P}$$

where

$$\begin{cases} size(v) = 1 & \text{for } v \in (N \cup T), \\ size(f^k(\bar{x})) = size(\bar{x}) & \text{for } \kappa \in \{\cdot, \epsilon, |, ?, *, +\} \end{cases}$$

2. It is presumed that each non-terminal symbol has exactly one option, it was considered that $ALT = \frac{\#P}{\#N}$
3. The formula suggested by Power and Malloy (2004) was followed, which defines:

$$MCC = \sum_{(n \to \alpha) \in P} mccabe(\alpha)$$

where

$$\begin{cases} mccabe(v) = 0 & \text{for } v \in (N \cup T), \\ mccabe(f^k(\bar{x})) = 1 + mccabe(\bar{x}) & \text{for } \kappa \in \{|, ?, *, +\}, \\ mccabe(f^k(\bar{x})) = mccabe(\bar{x}) & \text{for } \kappa \in \{\cdot, \epsilon\} \end{cases}$$

- **Structural metrics** that measure the dependency among the symbols of a grammar induced by its productions.

  Once again, we can state that the more intricate are the interrelations among the symbols, the harder it is to support the grammar and to recognize the sentences of the generated language. To compute those metrics, a grammar is represented as a graph.

Metric	Definition
#R	Number of recursive symbols
FanIn[1]	Average number of branches of the input nodes (non-terminals) of the DGS
FanOut[2]	Average number of branches of the output nodes of the DGS
TIMP[3]	Tree Impurity
CLEV[4]	Normalized Counts of Levels
NSLEV[5]	Number of Non-Singleton Levels
DEP[6]	Size of The Largest Level

Table 2: Metrics for evaluating the Structure of Context-Free Grammars.

Notes:

1. To calculate the average number of branches of the input nodes (non-terminals) of the DGS, it was assumed that $FanIn = \frac{\sum_{(n \to \alpha) \in P} size(\alpha)}{\#N} * 100$
2. To calculate the average number of branches of the output nodes of the DGS, it was considered that $FanOut = \frac{\sum_{(n \to \alpha) \in P} size(\alpha)}{\#N + \#T} * 100$
3. For a call graph, the tree impurity metric may be calculated using this formula

$$TIMP = \frac{edges - nodes + 1}{(nodes - 1) * (nodes - 1)} * 100$$

where

$$\begin{cases} nodes = \#N, \\ edges = \#\{(A \rhd^* B) | A, B \in N\} \end{cases}$$

4. To compute the normalized counts of levels, the following formula is applied $CLEV = \frac{\#(N_{\equiv})}{\#N} * 100$
5. The number of non-singleton levels is obtained by $NSLEV = \#\{n \in N_{\equiv} | \#n > 1\}$
6. To determine the size of the largest level, applies $DEP = \max\{\#n | n \in N_{\equiv}\}$

2.3 Language Complexity

Software security is turning into an inexorably significant differentiator for IT organizations. Therefore, methods for forestalling software vulnerabilities during software development are turning out to have increasing significance. The longer it takes to find the vulnerabilities, the more costly it will be to fix, and making an already difficult situation even worse.

In order to identify existing vulnerabilities, Static Application Security Testing, abbreviated as SAST and often alluded to as "White-Box Testing," is used. The tool performs a security test that examines the source code of applications.

The idea behind this type of analysis is to identify in the code the use of language constructions that are vulnerable and can facilitate external attacks on the SW system. Static analysis examines the text of a program statically, without running it.

2.3.1 *Checkmarx CxSAST*

The principal product of Checkmarx is a SAST tool, CxSAST, that analyses source code written in various programming languages and finds its security vulnerabilities (Li, 2020; Checkmarx, 2021a,b).

The peculiarity that distinguishes this tool from others is its versatility, as it supports almost all the most widely used programming languages in the market. To differentiate itself from the competition, Checkmarx CxSAST uses a special lexical analysis method and CxQL (Checkmarx Query Language) – a patented query technology to search for vulnerabilities.

In Figure 1, a high-level description of the scanning pipeline that allows for the aforementioned static analysis of the source code will be presented.

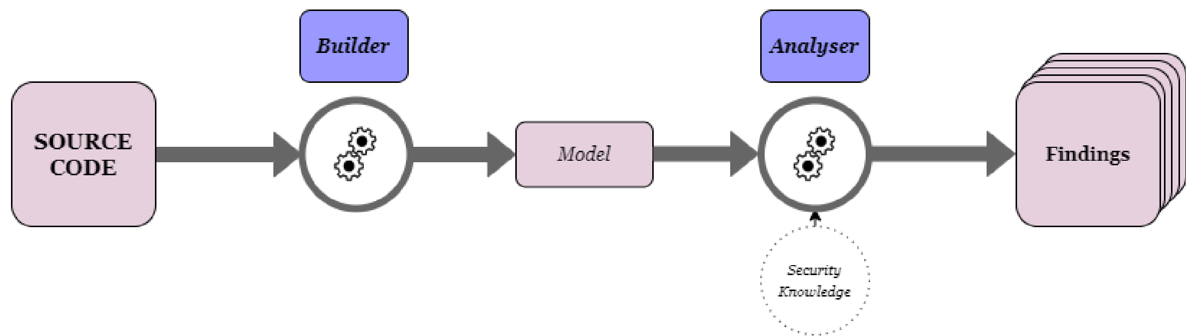


Figure 1: An abstract, high-level model of a SAST tool.

In a comprehensive way, a SAST tool basically comprises two processes.

1. Model **Builder** by ingesting the source code and transforming it into a normalized, understandable model for the analyser to decipher. A typical model is the development of an AST, a simplified representation of the structure of the source code, where each node in the tree is associated with a constructor of the code. In this way, the syntax is abstract since it does not represent every detail, as it appears in real syntax.

2. **Analyser based on existing security knowledge**, using a series of rules to figure out what the tool should evaluate within the source code. It is critical to customize and calibrate these rules to suit a specific application, as this allows for more reliable and worthwhile results. The feasibility of these methods is the main inner component of a SAST tool, from a customer's point of view.

The analyser based on current security knowledge process is irrelevant to the goal project since the challenge of supporting a new programming language resides in the model's builder process, because the language in which the source code is produced influences the static analysis process.

Any SAST tool can be described by Figure 1. Checkmarx CxSAST was the chosen analysis tool, so it makes sense to dig deeper into what steps are involved in building the model, what model is chosen, and what approach they take to the analysis.

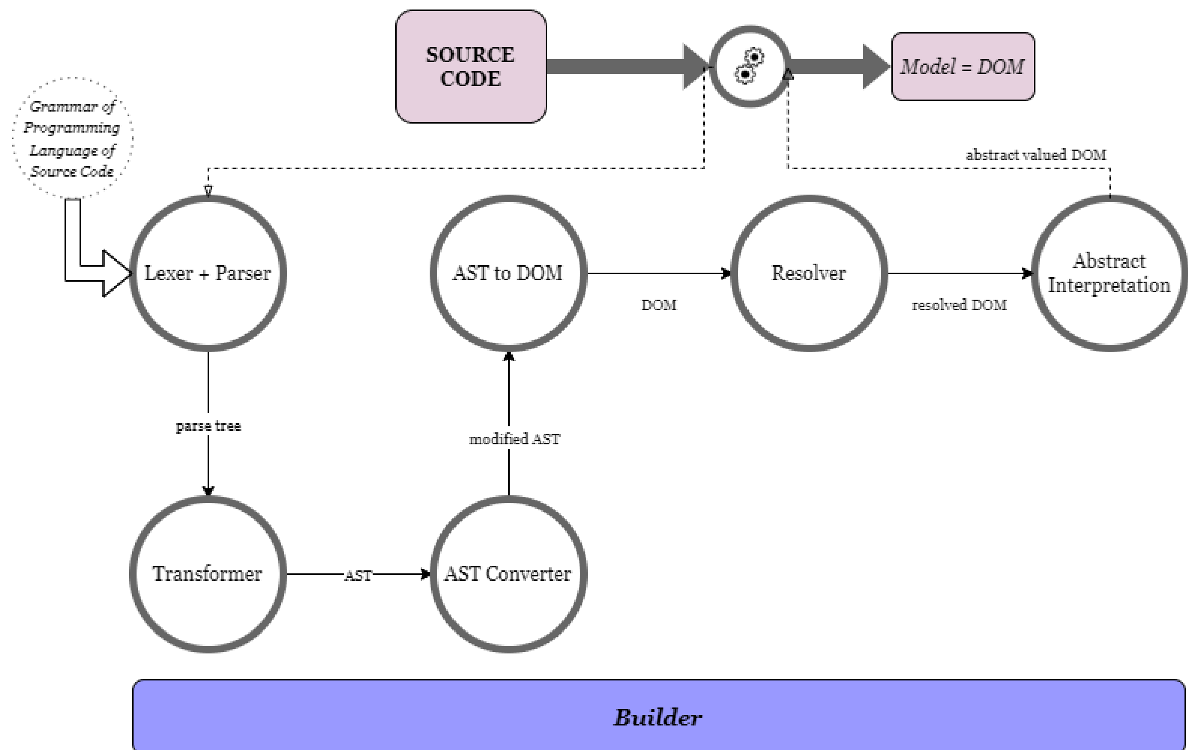


Figure 2: An abstract, high-level model of the SAST tool builder.

Next, a brief description of each phase of the Builder process, Figure 2, is presented.

- **Language Recognizer**: starting from the grammar of the programming language used in the source code under analysis, two steps are performed:
  - *Lexical analysis* phase, produced by a component usually called a Lexer, where the source code under evaluation is read and divided into tokens using lexical rules with the goal of identifying terminal symbols and specifying tokens. For this, these rules are defined using regular expressions. As an output of this phase, the Lexer produces a sequence of tokens for the next phase.
  - *The syntactic analysis* step is performed by a component typically called a parser, where it provides a structure for the token stream. For this, the order of the tokens is checked against the structural rules that define the structure and order of the token combination. As an output of this phase, the parser returns a parse tree.

  Creating a lexer and/or parser can be a very problematic and extensive process. To simplify lexical or static analysis, a parser generator called ANTLR (Parr, 2013) that uses the LL(*)[3] parser is used. The advantage of using this generator is that it provides a single, consistent notation for specifying the Lexer and Parser.

3 LL(*) means top-down parser from left to right, which constructs a Leftmost derivation of the Input and looks ahead to any number of tokens when selecting among alternative rules.

- **Transformer**: performed by a component, which follows a strategy or pattern of visitors, listeners, etc. The Visitor strategy lets us choose how to navigate the tree and which nodes we will visit. It likewise permits us to characterize how often we visit a node.

  In this phase, the parse tree is traversed in order to create some output. This traversal features an action for each node in the parse tree, where each action can give rise to a string or other complex object. As a result, an AST is generated.

- **AST Converter**: are required, regardless of the programming language the source code is in. Each of these conversions is characterized by a predicate (a condition) and an action (from an AST node, a function is created that returns a transformed AST node). The predicate of each conversion is tested against it and, perchance, the associated action is performed. This is true for each AST node constructed by the visitors.

- **AST to DOM**: mapping the AST nodes to a cross-language code representation – the DOM – which will be essential for all the next steps.

- **Resolver**: performs traversals on the DOM to complement some notions that have not been inferred up to this point. For example, the use of a variable or method in this phase, will look for its declaration and complement the node in the DOM with the identification of the declaration found.

- **Abstract Interpretation**: in this last step, the DOM is again traversed and analysed. However, in this phase, the goal is to infer the values that each variable can have, the potential values that each method invocation can return, the variable types, and other relevant information.

It should be emphasized that each step of model building depends on the complete completion of the previous step for the model to be correct and usable. Another point to note is that, in traversing the AST or DOM, the traversals through these structures can sometimes not be performed in parallel because of dependencies between the nodes.

Once the model building process is finished, the step of performing analysis uses the produced model, which is basically a populated data structure. CxSAST identifies vulnerabilities by querying this document object model (DOM).

2.3.2 *Supporting of Programming Languages by CxSAST*

In order to assess language complexity, it is necessary to identify factors that make the support process more time-consuming. Previously, in Figure 2 presented the phases required to build a model that enables the detection of vulnerabilities.

In this model construction, there is a step that transforms the source code written in a language to a representation independent of the programming language it is in (AST to DOM). For this reason, the complexity of the language only affects the steps that precede this transformation, those being the language recognizer, the transformer, and the AST converter.

In this sense, the complexity of supporting a new language in the context defined is strongly related to the AST (Parr, 2009). For this reason, for the support to be more productive, at the level of less time and effort, the programming language in which the code under analysis is written, should generate an AST with the following characteristics:

- **Dense**: No pointless nodes;
- **Convenient**: Patterns in the tree are straightforward and quick to discern;
- **Significant**: Emphasize operators, operands, and the connection between them.

The initial two focuses infer that it ought to be simple and quick to identify patterns in the tree. In order to gather as much useful information as possible about each node, it is usually necessary to make multiple passes over the tree. The last point infers that the tree structure ought to be insensitive to changes in the grammar.

Besides this strong relationship, there is another aspect that the complexity of the programming language influences, since when the tree is generated there are properties of the nodes that have not been assessed so far and are necessary in order to detect as many vulnerabilities as possible.

In general, it is intended that each node of the AST contains the following properties:

- **Name**: Symbols are identifiers like x and y, yet they can be operators as well, as for example, the symbol known as the addition operator (+);
- **Category**: The thing like the symbol is. Is it a class, method, variable, name, etc. To approve a method called x + y, for instance, we want to realize that + is a method to add 2 or more values, not a variable or class.
- **Type**: When the symbol is a variable, there is an interest in knowing what type it concerns, in order to be able to subsequently approve certain operations. Normally, the software engineer needs to explicitly distinguish between each type of symbol (in some languages, the compiler assumes this).
- **Scope**: The scope of a symbol restricting is the part of a program where the symbol is valid, that is, the place where the symbol can be utilized to refer to the element.

The cost of supporting a programming language in a SAST tool is a function of many of its properties. The following are example of features present in real programming languages that influence the complexity of support in this type of application.

- **Declaration** in a programming language is a statement that determines the properties of a symbol. A declaration introduces a program Entity identified by a unique name (an identifier), its category (function, variable, constant, etc.), its type (in case it is a variable or constant) or if it is a function what it accepts as input and output, it can also declare things like the size of a type.

  Example 5. *Consider for example the same code written in different programming languages, Python, Java, JavaScript, respectively, to declare the same variables and the same function.*

```
def main():
    x = 13
    y = "Python!"
```

```
public class Main {
    public static void main(String[] args) {
        int x = 13;
        String y = "Java!";
    }
}
```

```
function main() {
    let x = 13;
    var y = "JavaScript!"
}
```

A declaration conveys the "meaning" of a symbol, which highlights that this property is related to semantics and not syntax. However, there is interest in analysing this because of the extra effort needed in the Builder process.

The fact that a declaration is used to communicate the presence of an entity to the compiler means that, in dynamically typed languages such as Python, it is unnecessary to specify the variables; the runtime interpreter does the verification work. Considering the present case study, as shown, a static analysis tool does not execute any code. For that reason, in order to detect as many vulnerabilities as possible, it needs to do the work of the compiler.

In comparison, for example, to the extreme that a language that is strictly typed like Java, C# or C++ explicitly defines the data type when creating a symbol, which makes vulnerability detection easier than most things will be known, allowing reasoning and analysis with confidence.

There is also the case of the JavaScript language, where the variable declaration is dynamic but allows you to write using explicit types, getting the advantages of a strongly typed language. This makes the static analysis of vulnerabilities easier than dynamically modifying them.

- **Indentation** is not a property present in most languages, since its use is intended for the better understanding of programs, in particular in connecting the control flow built through conditions or cycles.

  This flow is controlled normally by the use of the symbols { }, which allows creating blocks where symbols can be used. Most languages consider white space as something disposable only for symbol division. However, in the Python language, white space is central to determining the structure of a program.

  Example 6. *Consider for example the same code written in different programming languages, Java, Python, respectively, to declare the same code.*

```
for (i = 0; i < 13; i++)
{if (i % 2 == 0)
    {doExample(i);}
 else
    {doAnotherExample(i); }}

```

```
for i in range(13):
    if i % 2 == 0:
        do_example(i)
    else:
        do_another_example(i)
```

  As you can see, the code presented in the second example is much clearer and more understandable. If we are only interested in understanding the code, indenting according to the block it belongs to is the best option, discarding the use of { }. However, for the detection of vulnerabilities, it is necessary to group the statements by blocks. For this, we need a token that allows the sign of the beginning and end of the same.

Considering programming languages that require the use of these limiters, there is no extra effort in determining the blocks. Otherwise, the lexical analyser needs to do some prior work because it needs to count the indentation, detect if the block is opening or closing, or if it is the same. In the above example from the Python language, by executing the call *do_example()*, the block inside the if is ended.

- **String literal** is one of the most common types in a representation language for representing a sequence of characters in the source code of a program. Typically, in modern programming languages, it is denoted by the beginning and end of quotation marks. However, there are several notations for declaring string literals, particularly more complex cases, in newer versions of programming languages. One such case is the specification that follows in the example below.

Example 7. *Consider the following example using Python f-string expressions.*

```
1   example = 1
    number_of_example = 6
3
    print(f'This is {example} example from the {example + number_of_example} examples presented so far')
5   """Equal to : This is 1 example from the 7 examples presented so far """
```

In the example above, it is shown that with this property, complexity can also come from the Lexer, although the most natural thing is to associate complexity with the parser. It is usually associated with the fact that parsers are so complex when they are too long.

Nevertheless, for the recognition process, size is not so relevant compared to grouping the characters into tokens because the Lexer needs syntactic context to decide, or as it is in this case, there are regions in the source code with different lexical rules.

- **Semicolon insertion** is used to separate and end statements. In a programming language that allows the use of the semicolon as a separator allows you to construct over one instruction on the same line, its use as an instruction termination shows to the compiler or interpreter where that instruction ends and the next one begins.

  In programming languages that do not require the use of semicolons, which have a great impact on a static analysis tool because no execution of the source code occurs, there is an effort to find out where an instruction begins and ends.

Example 8. *Consider, for example, the same code written in different programming languages, Java, JavaScript, respectively, to declare the same code, but where the second language does not require the use of a semicolon.*

```
a = b + c
(d + e).example();
// Equivalent to :
// a = b + c(d + e).example()
a = b + c;
(d + e).foo();
// Equivalent to :
// a = b + c
// (d + e).example()
```

```
a = b + c ;
(d + e).example();

```

As is shown, the absence of the semicolon brings ambiguity to the moment of analysis, hence its contribution to the increased complexity of the support. A concept also associated with this property is line continuation, where the Lexer needs to discard the backslash and the newline, rather than the newline being tokenized, as in most languages.

Example 9. *Consider, for example, the same code written in previously, but now in Python to demonstrate the use of line continuation.*

```
a = b + c \
(d + e).example()
```

- **Dynamic Memory Allocation** permits programs to execute with greater flexibility when the data on which they operate varies dramatically at run-time.

  Example 10. *Consider the following C code, which demonstrates a basic program that uses snprintf() to copy the number of command-line parameters and the name of the binary that was executed into the stack buffer str.*

```
int main(int argc, char **argv) {
    char *str;
    int len;
    if ((str = (char *)malloc(BUFSIZE)) == NULL) {
        return FAILURE_MEMORY;
    }
    len = snprintf(str, BUFSIZE, "%s(%d)", argv[0], argc);
    if (len >= BUFSIZE) {
        free(str);
        if ((str = (char *)malloc(len + 1)) == NULL) {
            return FAILURE_MEMORY;
        }
        snprintf(str, len + 1, "%s(%d)", argv[0], argc);
    }
    printf("%s\n", str);
    free(str);
    str = NULL;
    return SUCCESS;
}
```

Because the decision is deferred until execution, the increased complexity that dynamic allocation entails is obvious. Because objects are dynamically allocated and can be created and released at any time, in any order, discovering source code vulnerabilities is not an easy task. This implies for give support of this kinda of language adding code to determine the appropriate buffer size, allocating the new memory, and confirming that the allocation is correct. A language that uses a static allocation strategy, where the program is run the operating system reads and creates a process, the program in this case being a static notion and the process the running program, for vulnerability detection does not require the extra effort of understanding how memory is managed.

Taking this into consideration, the complexity of a language is defined by the complexity of the grammatical structure and the semantic properties — these two factors thus impact on the time and effort needed for support.

3

PROPOSAL

Although there are some similar tools, such as the *SynQ* tool (Power and Malloy, 2000, 2004), *SdfMetz* or *SdfCoverage* (Alves and Visser, 2005, 2008) and the *gMetrics* tool (Cervelle et al., 2009; Crepinsek et al., 2010) or even the *GQE* tool (Cruz, 2015), that provide information about the complexity or quality of a grammar, the tool that will be proposed in this chapter, called Language Complexity Evaluator (LCE), differs in its ultimate goal, to measure the complexity of a language rather than of the grammar.

However, the referred tools already demonstrated that grammar metrics are important for understanding the complexity of a grammar, therefore their implementation. Furthermore, LCE will bring innovation, as it allows for an evaluation of linguistic properties through a new DSL.

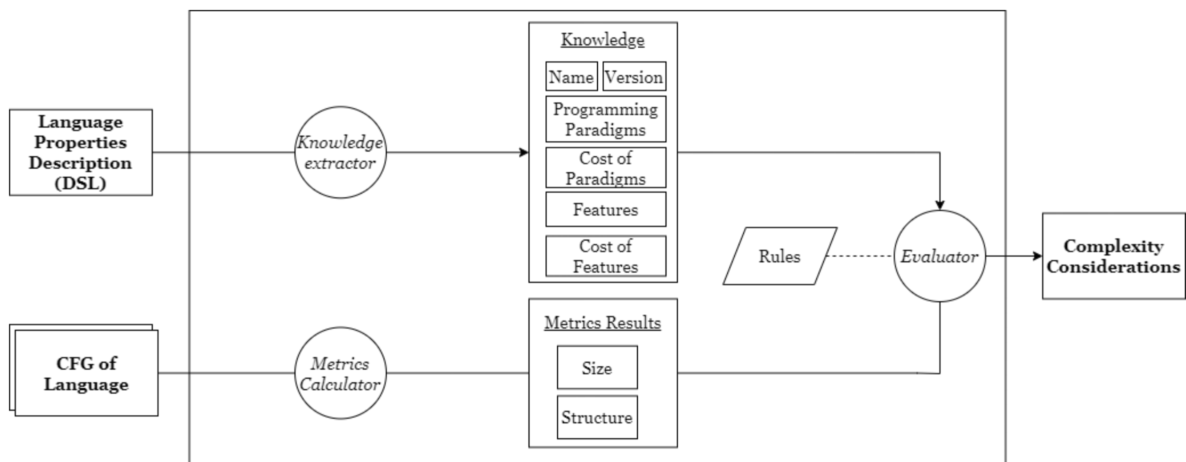


Figure 3: Language Complexity Evaluator, architecture.

The diagram that sketches the architecture of the tool under development is depicted in Figure 3.

The tool, LCE, is an application that reads any grammar and any language properties description, generating values for each of the metrics under consideration and extracting knowledge from the properties provided, respectively.

Using the parameters that measure the grammar complexity, so far computed, and weighting the features extracted from the language properties described, the last module evaluates the language complexity and reports the results achieved.

3.1 A DSL TO DESCRIBE THE PROPERTIES OF A PROGRAMMING LANGUAGE

As discussed in the previous chapters, there are aspects of a programming language that grammar metrics cannot measure or capture, but which have a huge impact on the support, particularly when it comes to the adaptation of a static analysis tool.

In this sense, a domain-specific language (DSL) called PropLD (Properties of a Language Description) was created with the goal of describing the features of a programming language that bring positive or negative impact to the moment of its support. PropLD is used for programming languages to define the substance of linguistic features and paradigms.

This DSL was not created with the intention of acting as documentation for a specific language, but its design enable the identification of various characteristics and notions that allow for an assessment of its complexity.

A visual representation of PropLD syntax is presented in Figure 4.

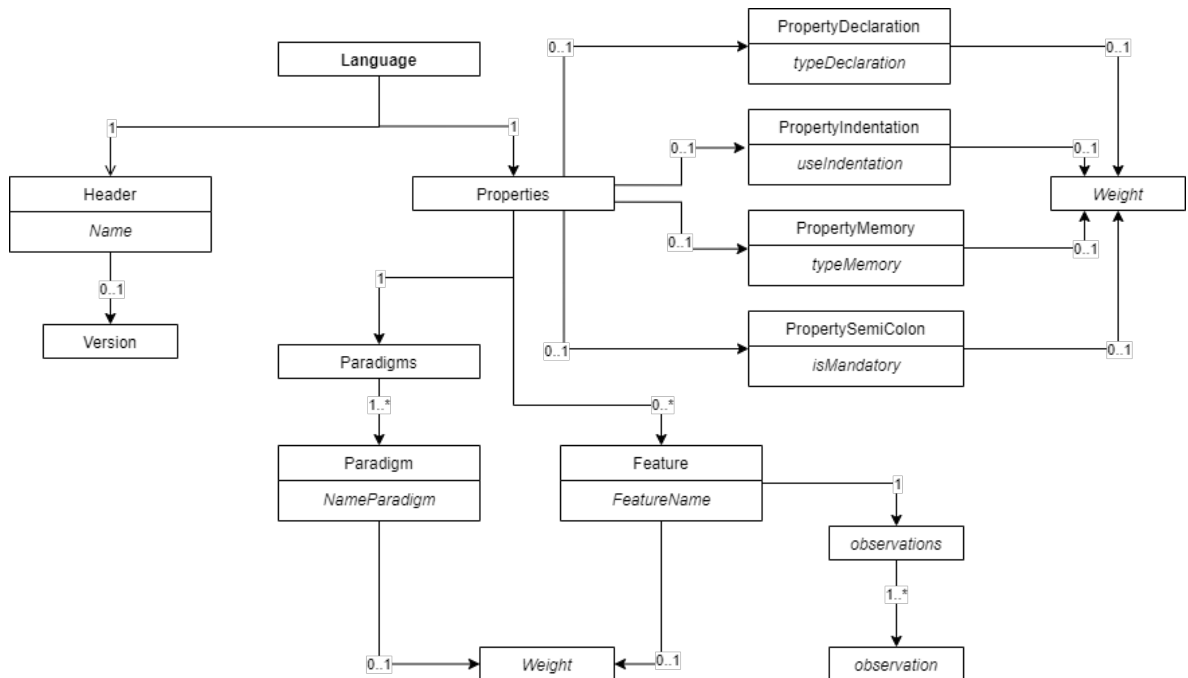


Figure 4: PropLD Syntax Diagram.

3.1.1 *Parser Rules*

The grammar that formally defines this DSL was written according to the ANTLR metalanguage (for more information about this tool, please read Appendix A).

The grammar rules used to support the automatic generation (resorting to ANTLR) of a parser to recognize the DSL to describe the Properties of a Language are shown in Listing 3.1. The lexer rules (used to generate automatically the Lexer Analyser) that specify the Terminal symbols used in the syntactic rules below are listed in Appendix B.

```
1  grammar LanguageProperties;

3  language: header properties;

5  header: name (version)?;

7  name : ID;

9  version : VERSION;

11 properties: paradigms
       (SEMICOLON propertyDeclaration)?
13     (SEMICOLON propertyIndentation)?
       (SEMICOLON propertyMemory)?
15     (SEMICOLON propertySemiColon)?
       (SEMICOLON feature (SEMICOLON feature)*)?
17     DOT;

19 paradigms: PARADIGM COLON paradigm (COMMA paradigm)*;

21 paradigm: nameParadigm (weight)?;

23 nameParadigm: ACTION | AGENTORIENTED | ARRAYORIENTED
       | AUTOMATABASED | CONCURRENTCOMPUTING | CHOREOGRAPHICPROGRAMMING
25     | RELATIVISTICPROGRAMMING | DATADRIVEN | DECLARATIVE
       | FUNCTIONAL | FUNCTIONALLOGIC | PURELYFUNCTIONAL
27   | LOGIC | ABDUCTIVELOGIC | ANSWERSET
     | CONCURRENTLOGIC | FUNCTIONALLOGIC | INDUCTIVELOGIC
29   | CONSTRAINT | CONSTRAINTLOGIC | CONCURRENTCONSTRAINTLOGIC
     | DATAFLOW | FLOWBASED | REACTIVE
31   | FUNCTIONALREACTIVE | ONTOLOGY | QUERYLANGUAGE
     | DIFFERENTIABLE | DYNAMICSCRIPTING | EVENTDRIVEN
33   | FUNCTIONLEVEL | POINTFREESTYLE | CONCATENATIVE
     | GENERIC | IMPERATIVE | PROCEDURAL
35   | OBJECTORIENTED | POLYMORPHIC | INTENTIONAL
     | LANGUAGEORIENTED | DOMAINSPECIFIC | LITERATE
37   | NATURALLANGUAGEPROGRAMMING | METAPROGRAMMING | AUTOMATIC
```

```
    | INDUCTIVEPROGRAMMING | REFLECTIVE | ATTRIBUTEORIENTED
39  | MACRO | TEMPLATE | NONSTRUCTURED
    | ARRAY | NONDETERMINISTIC | PARALLELCOMPUTING
41  | PROCESSORIENTED | PROBABILISTIC | QUANTUM
    | SETTHEORETIC | STACKBASED | STRUCTURED
43  | BLOCKSTRUCTURED | STRUCTUREDCONCURRENCY | OBJECTORIENTED
    | ACTORBASED | CLASSBASED | CONCURRENT
45  | PROTOTYPEBASED | BYSEPARATIONOFCONCERNS | ASPECTORIENTED
    | ROLEORIENTED | SUBJECTORIENTED | RECURSIVE
47  | SYMBOLIC | VALUELEVEL | OTHER;

49 weight: DOLLAR NUM;

51 propertyDeclaration: DECLARATION COLON typeDeclaration (weight)?;

53 typeDeclaration: STATIC | DYNAMIC | BOTH;

55 propertyIndentation: INDENTATION COLON useIndentation (weight)?;

57 useIndentation: YES | NO;

59 propertyMemory: MEMORY COLON typeMemory (weight)?;

61 typeMemory: MANUAL | AUTOMATIC;

63 propertySemiColon: STRSEMICOLON COLON isMandatory (weight)?;

65 isMandatory: YES | NO | OPTIONAL;

67 feature: featureName COLON LPAR observations RPAR (weight)?;

69 featureName : STR;

71 observations: observation (COMMA observation)*;

73 observation: STR;
```

Listing 3.1: Syntatic Rules of the PropLD.

The sentences written in PropLD express precisely the features of a programming language under analysis. The language was designed in such a way that Terminal symbols express literally the individuals they denote. As can be observed in Listing 3.1, most of the concepts (represented by Non-terminal symbols) have alternatives to described directly the possible choices for that concept. As a consequence, PropLD is easy to learn, to use and its sentences are clear and simple of understanding.

3.1.2 *Example*

To clarify the design of PropLD DSL, as well as to demonstrate its use, Example 11 illustrates how to describe the features of a real programming language.

Example 11. *Description of the properties of the Python programming language, written in PropLD DSL, introduced in the previous section.*

```
Python 3.10.4

PARADIGMS : object-oriented $3, procedural, functional,
    structured, reflective;

DECLARATION : dynamic ;

INDENTATION : yes $-1 ;

MEMORY : automatic $1 ;

SEMI-COLON : no ;

"Anonymous functions" : [
    "Implemented using lambda expressions;
        however, there may be only one expression in each body"
];
"String interpolation": [
    "The process of evaluating a string literal containing 1..*
        placeholders, yielding a result in which the placeholders are
            replaced with their corresponding values."
] $20;
"Triple-quoted": [
    "Beginning and ending with three single or double quote marks.",
    "May span multiple lines and function like documents in shells"
]
.
```

4

LANGUAGE COMPLEXITY EVALUATOR

After presenting the concepts involved in this project, as well as, an explanation of the proposal of the tool to achieve the desired objectives, the LCE is presented, a tool that allows the evaluation of the complexity of a language based on a set of metrics to classify the complexity of its grammar, along with a set of features available as an open source project at `https://lce.di.uminho.pt/`.

4.1 ARCHITECTURE AND TECHNOLOGIES

This section presents the technologies used to implement the architecture introduced in the previous chapter, and how they interact with each other in each user execution. LCE is more than a basic interface; it is composed of three components: frontend, backend, and database, as shown in Figure 5.

The major component, the frontend, is meant to offer the interface through which the user will interact with and evaluate his languages. The backend exists primarily to realize the process of calculating metrics and extracting knowledge, but it also provides certain additional queries and improves communication between the frontend and the database.

The database is used to store the previous languages already analysed for long-term storage, the information stored and used in the comparison against the metrics calculated, and the knowledge extracted.

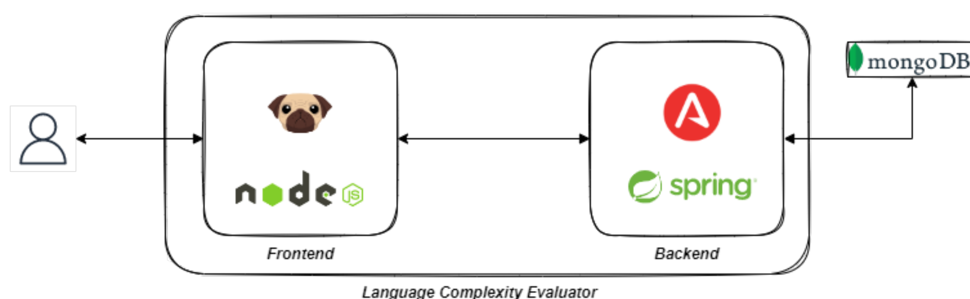


Figure 5: Diagram showing LCE architecture and technologies used.

The frontend was created using Express and Pug technologies, which are Node.js frameworks. The Pug framework was chosen because it is a well-known and feature-rich high-performance JavaScript template engine for this environment, acting as a bridge between Node.js and HTML by, for example, replacing application variables with current values at runtime and sending the resulting HTML sequence to the client.

The advantages of this choice are obvious: it offers a simple and effective support to create and maintain a great, easy-to-use web interface.

The open source framework Spring was utilized to construct the application's backend, which is written in the Java programming language. This language was chosen because it is a free object-oriented language with multiples libraries, making development easier, and it is one with which the author is familiar.

Spring was chosen as the framework because it provides numerous modules that may be employed depending on the demands of the project, including modules for web development, persistence, remote access, and object-oriented programming.

Furthermore, because this application should be able to read a grammar written in a certain meta-language and parse sentences of a DSL created by the author thought the said grammar, as well as take into consideration the programming language used, ANTLR, a parser generator previously presented, has been integrated.

MongoDB was the database chosen to store all information, which is a NoSQL document database that uses JSON-like documents to preserve data.

In this manner, the user provides any ANTLR grammar (that is, any grammar written in the ANTLR metalanguage) and any text written in the author's metalanguage into the application's frontend, which transfers the information from these files to the backend.

It is the responsibility of this component to create values for each of the metrics under analysis and extract information from the specified attributes, storing everything that has been computed and inferred in the database.

Through this architecture and technologies used, the user has the ability to analyse these values, as well as, observe some automatically generated considerations from a series of predefined rules. In this way, the user can easily predict whether the language is of low or high complexity.

4.2 BACKEND

This section begins with an explanation on how the backend was created from a more abstract perspective, then move to displaying the internal structure as well as its components, and how they all interact with each other. The underlying algorithms will also be discussed.

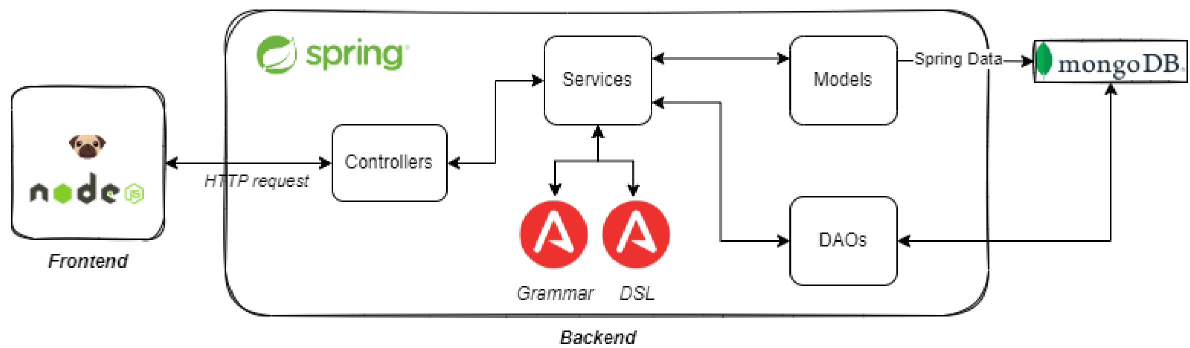


Figure 6: Backend structural diagram.

The backend functions as a data service, exchanging information via HTTP requests.

It is vital to note that this component was created using the Spring framework's Dependency Injection idea, in other words, to construct a specific class Y that would "assemble" the functionality of another class X. As a result, the X class must understand how to create and utilize the Y class, making them dependant.

In this sense, Dependency Injection is concerned with preserving the weak coupling between the classes and relieves one class of the obligation of knowing how to create the other.

The backend of the LCE tool is made up of six primary components, as shown in Figure 6.

4.2.1 *Data Models*

This subsection will describe the role of this component, how it was created, and the classes contents. The Java classes known as Models were developed to understand the ideas of context-free grammars and features of a programming language.

The primary purpose of this component is to store the results of a recognition of grammatical or linguistic properties and give the other components a way to manipulate the information.

The advantages of this approach are that it eliminates the entire grammar computation procedure, trying to make the language data simpler and easier to comprehend.

With the approach used, the program's simplicity of maintenance and autonomous development process are essential additional elements. For example, any update to the application may be made by modifying a value in the desired object.

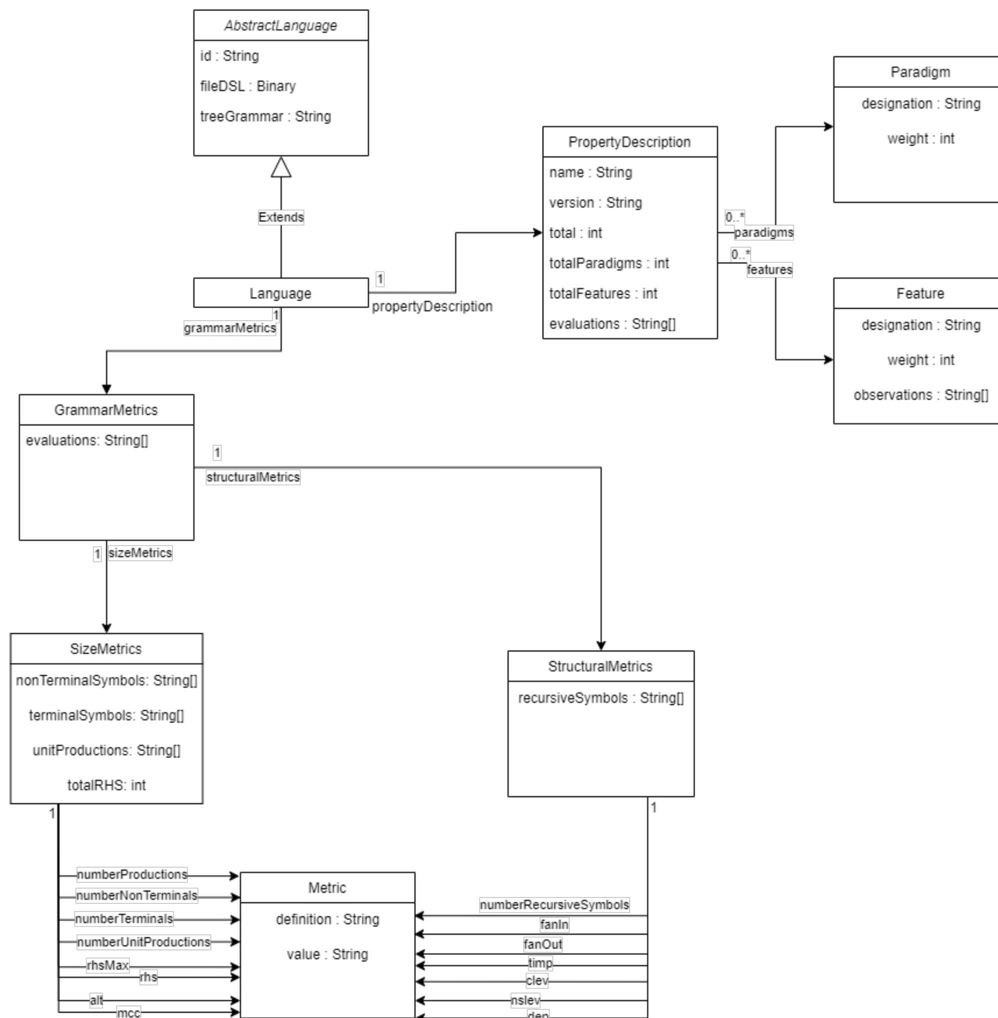


Figure 7: Diagram of the Data Models component's structure.

After describing the structural aspects of Data Models component, it is important to look at Figure 7, which displays the primary Java classes, each of which represents a different object.

The abstract class with the same name, *AbstractLanguage*, defines the component's top-level object. The benefits of this technique are not utilized in this project (yet), but they may be useful for a subsequent implementation, hence this primary object cannot be constructed.

The *Language class* was developed to contain all metrics and information gleaned through assessments of language complexity. The majority of parameters or, since they are implemented, do not pose any questions, but it is crucial to defend some choices.

- The created tree is kept in the *treeGrammar* variable in the grammar's JSON-formatted storage, which can be used to examine the added data. A more straightforward strategy was used for the description of linguistic qualities, and only the file's contents are saved.

- Both Terminals, Non-Terminals, Unit-Productions and Recursive-Symbols own a set to save their symbols, but there are also respective four variables of *Metric* type to store their values and definition since so many of the more metrics used the variables in their own definition; therefore, it is more efficient to only calculate the sets size once and store it in a value, for both sets, rather than doing so repeatedly;

In terms of the methods for each class, there are primarily two sets of methods implemented in this class: *getters*, which are required for exchanging data between components, and *setters*, that is used to save values in variables.

4.2.2 *DAOs*

As already said, the backend communicates with a database; in these tool, MongoDB. Regardless of the database type, this component must communicate through CRUD operations, enabling the creation, retrieval, updating, and deletion of items.

As a result, a class using the Data Access Object (DAO) pattern was developed, encapsulating database access and providing a channel for communication between the database and the various other backend components.

The application is not dependent on the implementation of a particular database type thanks to this abstraction and encapsulation of the database; for instance, if the database paradigm is changed in the future, the change will have very little effect. Additionally, the data layer is isolated from the layer that calculates metrics and extracts knowledge.

The code in Listing 4.1 displays the DAO interface that has been put into practice and provides various fundamental CRUD functions, some of which are helpful for presenting information to the user.

```
1 public interface LanguageDAO {
      boolean addLanguage(Language language);
3     Language getLanguage(String name, String version);
      Language getLanguageById(String id);
5     SizeMetrics getSizeMetrics(String id);
      StructuralMetrics getStructuralMetrics(String id);
7     PropertyDescription getPropertyDescription(String id);
      List<String> getEvaluationsGrammar(String id);
9     List<String> getEvaluationsPropertyDescription(String id);
      List<Language> getSimilarLanguages(String id);
11 }
```

Listing 4.1: DAO Interface

The class that implements the aforementioned interface is very simple and intuitive, and each of the method names precisely translates the purpose of each operation.

Additionally, each method's implementation is a typical MongoDB query to search for the desired data using the supplied parameters.

The procedure used to add a new programming language should be emphasized, though, as it was thought that the new information would take the place of the old one if there was already a programming language with the same version available in the database when the new one was added.

Algorithm 1 describes the procedure to add a new language in LCE using the DAO pattern follows:

Algorithm 1 Add Language

Input The language being evaluated
Output Boolean value to indicate whether the operation was successful or not

1: *check* ← check if exists a language with *language.name* and *language.version* provided
2: **if** *check* $\neq$ **null then**
3: update an *language* into the *collection*
4: **else**
5: insert an *language* into the *collection*
6: **end if**

4.2.3 *ANTLR*

ANTLR is a crucial component of backend structure, not so much because of its complexity, but rather because of what it does and what it stands for. The provision of all the ANTLR files to the other components, or more specifically to the services, is the primary duty of this component, as is obvious from the name of this component. Notice that this component actually encapsulates 2 of the 6 modules referred above when describing Figure 6.

The *ANTLRv4Parser.g4*, *ANTLRv4Lexer.g4* and *LexBasic.g4* files, which are accessible from this tool's public GitHub repository, were used to process a new grammar written in the ANTLR format. From the above files that are grammars to define ANTLR meta-language, two other files, a file with the *.tokens* extension and a file with the *.java* extension, were created using the ANTLR tool.

These files made it possible to build a lexer and parser object and carry out grammar parsing. The only changes between this process and the code in Listing A.1 are that the grammar start symbol in this process is **grammarSpec** and the input stream originates from an input file that was sent by the frontend.

When it comes to recognizing the description of linguistic properties written in PropLD language, the process was similar to the one mentioned above. In this case, PropLD grammar was written in the ANTLRv4 metalanguage format, which contains the parser rules (shown in 3.1) and the lexer rules (available in the appendix B). Afterwards, the same file generation process was performed using again the ANTLR tool.

Having the files created, it was already possible to use the generated lexer and parser to validate the language properties according to the specified syntax. This process is similar to the one shown in Listing A.1 where the initial grammar symbol is **language** and the input stream comes from the input file sent by the frontend

4.2.4 *Services*

Services component includes the elements responsible for the implementation of the business logic. The service objects instantiate the Java classes built to create, find, or calculate what is requested using the data models created with the help of the DAO object to access the database or the files generated by ANTLR to validate the input files.

To make the business logic process more explicit and limited, several services were created, where each one is responsible for performing operations on a given Data Model. In this sense, the following services were implemented:

- **StructuralMetricsService** - class responsible for evaluating the *structure metrics* of the grammar;
- **SizeMetricsService** - class responsible for evaluating the *size metrics* of the grammar;
- **PropertyDescriptionService** - object responsible for handling the *linguistic properties*;
- **GrammarService** - object responsible for processing the grammar. For this, it invokes the services that allow to evaluate the size and structure metrics, *SizeMetricsService* and *StructuralMetricsService*, respectively.
- **LanguageService** - main service that determines the complexity of a language. For this purpose it invokes the *GrammarService* and *PropertyDescriptionService* respectively. To store the information, as well as, extract the calculated information, it uses the DAO component that allows it to obtain the data present in the database.

4.2.5 *Controllers*

The controller component encapsulates the methods responsible for providing the Rest API. This architecture was selected because it enables the independence of the components without sacrificing the simplicity of access to the business logic, as was previously discussed for in Section 4.1.

With this approach, the frontend makes requests to these control methods, usually called endpoints, which get the appropriate response to the desired query.

When an endpoint is called, it begins handling the web request by communicating with the service layer (*LanguageService*) to carry out the necessary tasks.

This endpoint is then responsible for invoking the methods available in the service component, getting objects that it then returns to the frontend, which then presents the response to the user.

In this sense, the backend of LCE have these available endpoints with the respective HTTP method that allows to evaluate a new language or get information about some parameter of this evaluation (such as metrics, generated considerations, etc.).

- **POST /api/languages/newEvaluation** - add a new language to the system;
- **GET /api/languages/validateToken** - validate token presented in the request header;
- **GET /api/languages/structure** - get the grammar structure metrics results;
- **GET /api/languages/size** - get the grammar size metrics results;
- **GET /api/languages/language/{languageID}** - search the language with the identifier provided in path parameter;
- **GET /api/languages/knowledge** - get the properties' description;
- **GET /api/languages/graph** - get the dependency graph of symbols;
- **GET /api/languages/grammarTree** - get the tree of grammar that define the language present in the token;
- **GET /api/languages/evaluations** - get the considerations;
- **GET /api/languages/dslFile** - get the file of DSL of Linguist Properties provided;
- **GET /api/languages/compare** - get the list of languages with similar paradigms;
- **GET /api/languages/commonParadigms/{languageID}** - get the list of common paradigms for the language with the identifier supplied by path parameter;
- **GET /api/languages** - get the language;

Each endpoint shown above is the URL of the backend from which the frontend can access the resources needed to perform their function. As can be observed, most of the implemented endpoints should not raise any questions, as their URL is quite explicit or the associated description should help explain their function.

However, it may be warranted to explain why the endpoint *GET /api/languages/validateToken* was created, and consequently how the language identifier under evaluation is provided to the backend.

In the developed application, one of the goals was to recognize a grammar in the ANTLR format and the description of features in the previously presented DSL format.

To achieve this goal, it was decided to highlight the use of *JSON Web Tokens* (JWT) since it is an Internet standard for creating data with optional signature and/or cryptography whose *payload* contains a *JSON* object with information.

The idea behind this use is that the backend generates a *token* in order to identify the language to be evaluated, so the user can use this assigned *token* to prove that he is connected to the system and prove that he has provided valid files.

The *tokens* were designed to be compact, URL-safe and usable, especially in the context of a single web browser evaluation. JWT declarations were used to convey the identity of this language case under evaluation between the various implemented services.

4.3 FRONTEND

This component's goal is to implement the interface between the LCE and the user, and actually its main role is to output the results produced by the other component, the backend.

This service's goal is to display the results of the computed metrics and extract knowledge about the linguistic aspects of the language from the specified grammar and description in the DSL-format, respectively.

Although it is a very simple web application, it was designed to be efficient, easy to use by any user and with access to the results in a few clicks, and aesthetically simple.

To prove that the application does indeed comply with what was said, it will be presented some images that show how easy it is to interact with, and how the information is accessible in an intuitive and legible way.

Figure 8 shows the simple home page that is displayed when the user accesses LCE tool. That page requests the names of the files in readable formats.

Taking into account that possible users may not be fully aware of what is necessary to input to evaluate a language, a help page was created that intends to aid the user at this step (Figure 9), providing a description of how to write a grammar in the ANTLR metalanguage or how to specify the properties of a programming language in the correct format.

This component provides information in a static way (without communicating with the backend) but allows anyone to start using the tool that has been created.

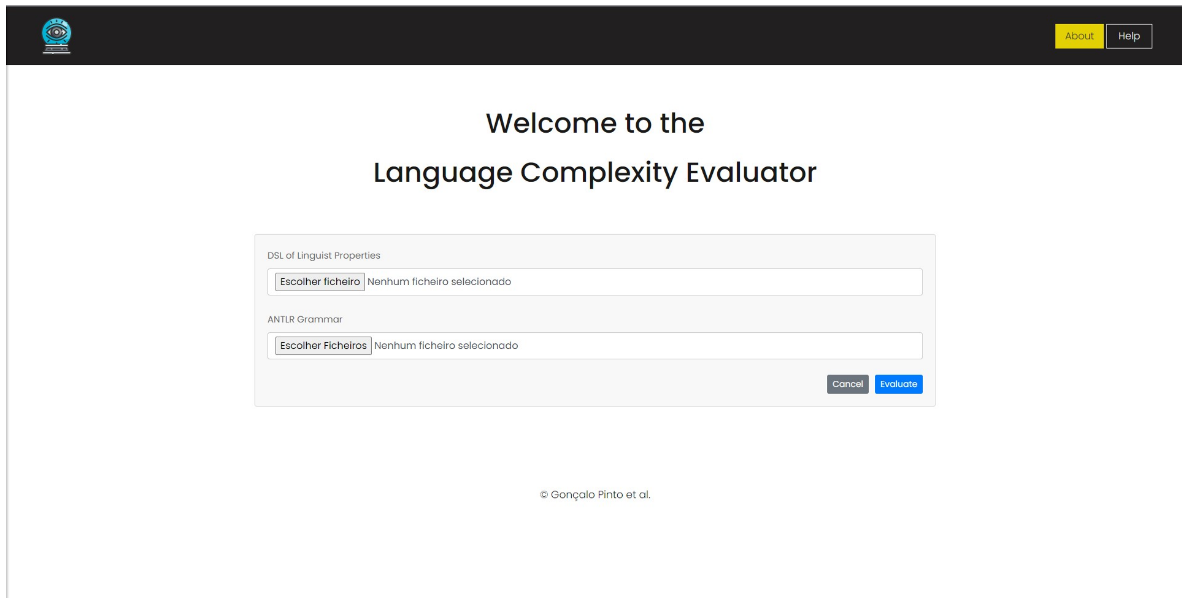


Figure 8: Main page.

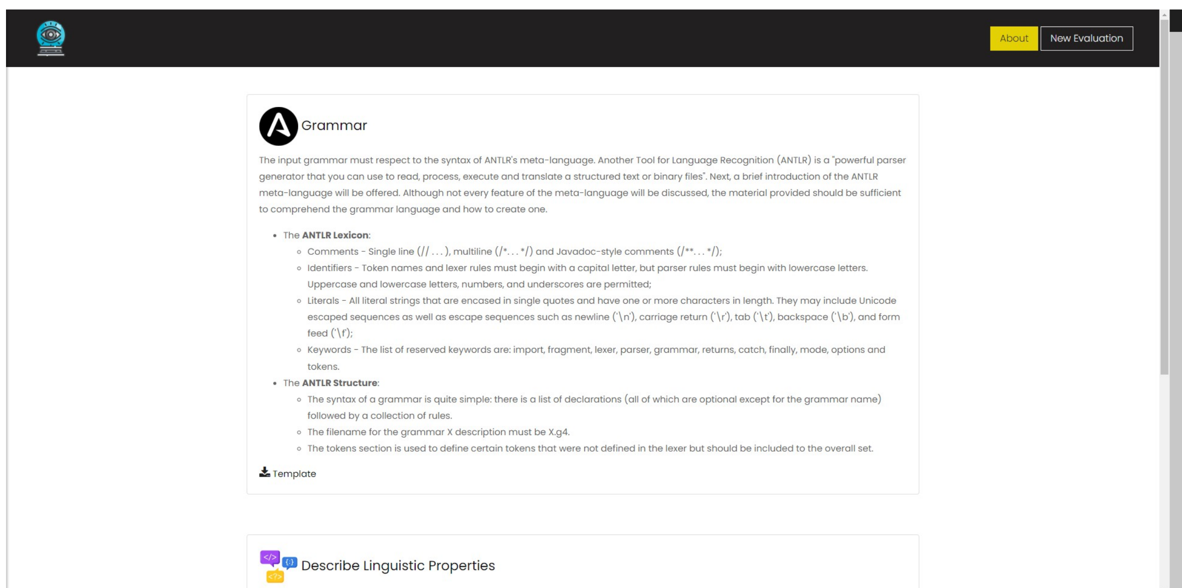


Figure 9: Help page.

In addition, as can be seen in the upper corner of each image previously mentioned, a way is provided for the user to have a context of the tool, as well as, how to contact the creators of the platform (details shown in Figure 10).

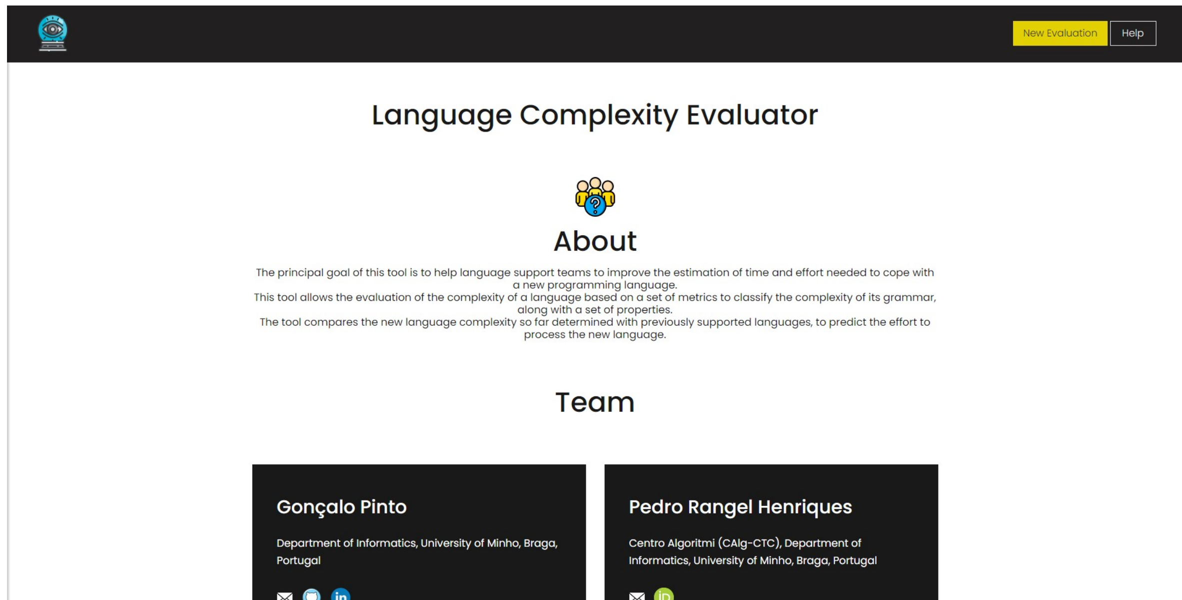


Figure 10: About page.

After entering a correct grammar, as well as a description of the language features, the user is forwarded to a new screen displayed in Figure 11, where the set of all possible operations is presented in a carousel format that, when selected, refers to information on another specific page.

However, one may not know that it is possible to get all of them from this carousel. So, a sidebar, was created to exhibit all the operations. To be easy to read, it was divided into different categories; in a first stage operation are hidden, but with a simple click the category label of all the concepts of that category is visible.

As can be seen in Figure 12, the operations associated with grammar have been expanded. However, not all of them are visible because there is a subcategory referring to metrics also compressed. To access the information, just click on the name present in this sidebar, causing the categories to be expanded or reduced; if an operation, is pressed, the user is directed to the respective page.

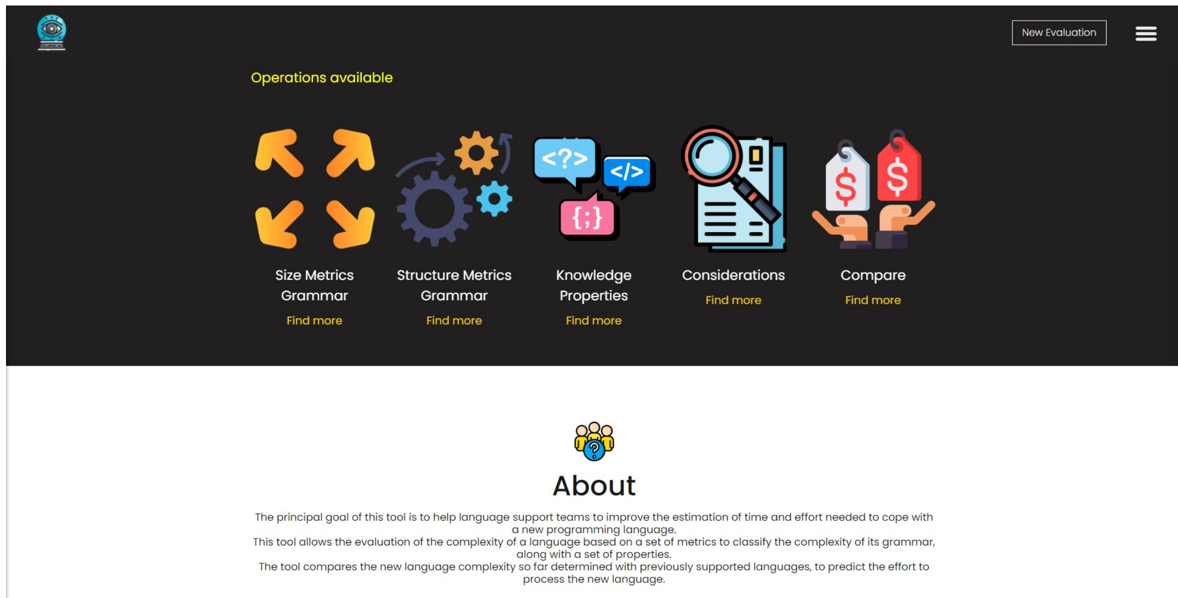


Figure 11: LCE operations available in format of a carousel.

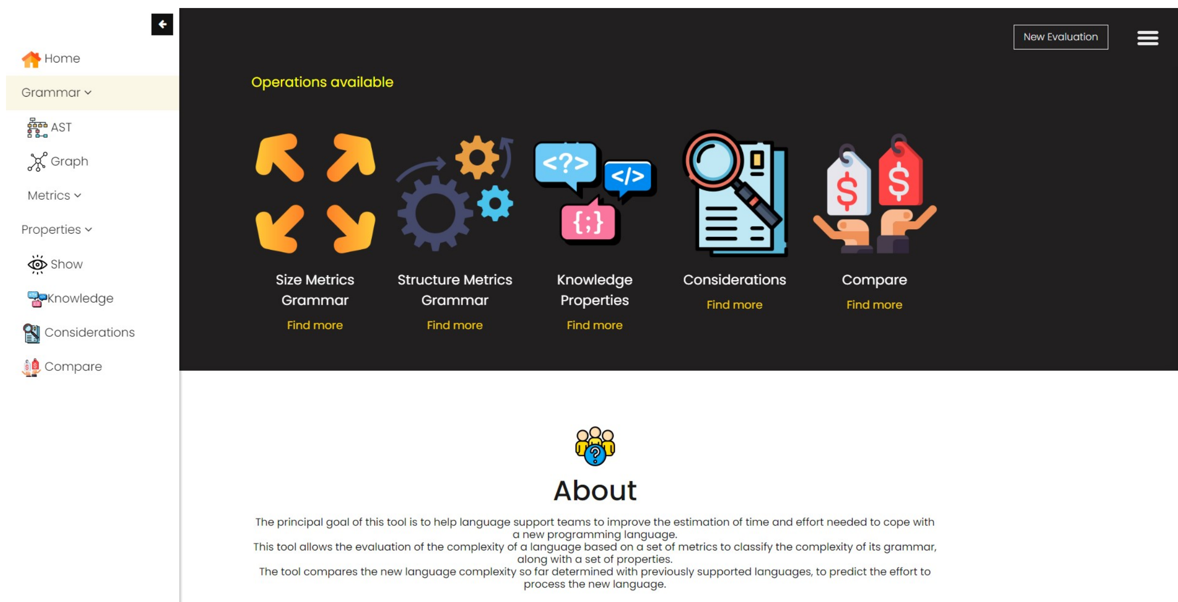


Figure 12: Operations available in format of a sidebar list.

LCE tool effectively performs automatically the entire procedure; it also includes visual representations of the Grammar Tree and the Dependency Graph between symbols (DGS), as can be seen in the Figures 13 and 14.

Figure 13: Visual representation of the grammar.

Figure 14: Visual representation of the DGS.

4.4 PREDICT EFFORT TO ADAPT CXSAST TOOL

As stated in Section 4.1, in order to predict the effort to process a programming language automatically, MongoDB, saves the metrics calculated, and the knowledge extracted from a language description each time the processing of a new language is performed. In this way, LCE is able to keep memory of all the languages previously evaluated.

This information comprises headers, paradigms, features, size metrics, and structure metrics. It is used as long-term storage, and allows comparing the new analysis results with the information stored in order to predict the effort.

Therefore, the user may simply compare the material supplied by the LCE tool to the languages already supported, aiming to forecast the work required to support a new programming language in its static analysis tool.

Algorithm 2 specifies how LCE finds and returns a list of programming languages previously analysed, that are similar to the one under evaluation.

Algorithm 2 Get Similar Languages

```
    Input The identifier of the language being evaluated
    Output List of Similar Languages
 1: result ← new list of similar languages
 2: language ← get language with given ID
 3: for all other language ∈ the database do
 4:     if other.version is not empty then
 5:         if other.name ≠ language.name and other.version ≠ language.version then
 6:             count ← count the number of equal paradigms
 7:             similarity ← (count/language.paradigms.size) ∗ 100
 8:             if count ≥ 50 then
 9:                 add other to result
10:             end if
11:         end if
12:     else
13:         if other.name ≠ language.name and other.version ≠ language.version then
14:             count ← count the number of equal paradigms
15:             similarity ← (count/language.paradigms.size) ∗ 100
16:             if count ≥ 50 then
17:                 add other to result
18:             end if
19:         end if
20:     end if
21: end for
```

5

CASE STUDY

This chapter seeks to demonstrate the real application (using as a case study a well-known programming language) of the LCE tool, described above, in its present state publicly available.

The purpose is to demonstrate the advantages of utilizing this tool to estimate the time and effort needed to handle a new programming language in a SAST tool.

To do this, an actual grammar of a programming language will be utilized as a case study to demonstrate the various outcomes offered by the tool.

Python is the programming language chosen as a case study. Python is a general-purpose programming language, adaptable, and powerful. Because it is brief and easy to read, it is a good first language. Python can be used to solve problems in almost all areas, from web development to machine learning.

The grammar that specifies the Python programming language, developed for ANTLR v4[1], and the description of the same programming language's features, shown in Example 11, are then considered as input files.

5.1 PROGRAMMING LANGUAGE ANALYSIS

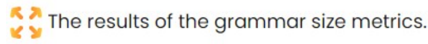
The outcome of the LCE evaluation of the complexity of Python will be reported in two stages: first the results of the grammar metrics are discussed, and then the knowledge extracted from the linguistic attributes supplied is presented. Finally, some conclusions concerning these two phases will be drawn.

- **Metrics Results**

  Figures 15 and 16 show the values computed for each grammatical metric previously defined in Tables 1 and 2.

1 The lexer and parser grammar can be found in the grammars-v4 repository of ANTLR.

Grammar Size Metrics



Metric	Description	Value
#P	Number of Productions	122
#N	Number of Non-Terminals	86
#T	Number of Terminals	89
#UP	Number of Unit Productions	6
RHS-MAX	Maximum number of symbols on an RHS	7
RHS	Average number of symbols in the RHS	3.877
ALT	Average Size Of Alternative Productions	1.419
MCC	McCabe cyclomatic complexity	2.721

Non-Terminal(s)

Search:

Symbol
and_expr
and_test
annassign

Showing 1 to 3 of 86 entries

Previous 1 2 3 4 5 ... 29 Next

Terminal(s)

Search:

Symbol
'%='
'&'
'&='

Showing 4 to 6 of 89 entries

Previous 1 2 3 4 5 ... 30 Next

Unit Productions

Search:

Symbol
break_stmt:'break';
continue_stmt:'continue';
encoding_decl:NAME;

Showing 1 to 3 of 6 entries

Previous 1 2 Next

Figure 15: Grammar size metrics for Python evaluated by LCE tool.

Grammar Structure Metrics

Metric	Description	Value
#R	Number of Recursive Symbols	48
FanIn	Average number of branches of the input nodes (non-terminals) of the DGS	5.5
FanOut	Average number of branches of the output nodes of the DGS	2.703
TIMP	Tree Impurity (%)	43.917
CLEV	Normalised Count of Levels (%)	46.51162790697674
NSLEV	Number of Non-Singleton Levels	2
DEP	Size of Largest Level	35

Recursive Symbols

Search:

Symbol
and_expr
and_test
arglist

Showing 1 to 3 of 48 entries Previous 1 2 3 4 5 ... 16 Next

Figure 16: Grammar structure for Python evaluated by LCE tool.

- **Knowledge**

  Example 11 offers a brief overview of some of Python's linguistic features. The information inferred by LCE from the language description provided (figure 17) are used to measure also the main characteristics of this language, as follows:

  – It includes five programming paradigms, one of which, object-oriented, has a weight of three units. The tool also creates the following assignments: the functional paradigm received a value of three units, the procedural paradigm received a value of two, and the others received a value of one (reflective and structured). This takes the overall weight associated with the paradigms to 10 units.

  – As for the linguistic features themselves, seven were indicated, and four of them did not have an associated weight. The tool assigned a support weight to these features (declaration type, whether it enforces the use of semicolons, Triple-quoted feature, and the anonymous functions feature).

Regarding the declaration type of variables and the use of semicolons, since both properties have a great impact on static vulnerability analysis, a weight of 9 and 10 units respectively was assigned. As for the other characteristics, a weight of 5 was assigned to each. This takes the overall weight attributed to the features to 49 units.

Knowledge Properties

Here are the extraction of knowledge about the linguistic properties.

Paradigms - total weight associated 10

Show entries Search:

Designation	Weight Associated
Paradigm functional	3
Paradigm object-oriented	3
Paradigm procedural	2
Paradigm reflective	1
Paradigm structured	1

Showing 1 to 5 of 5 entries Previous 1 Next

Features - total weight associated 49

Show 10 entries Search:

Designation	Weight Associated
Feature "String interpolation"	20
no semi-colon	10
dynamic declaration	9
Feature "Triple-quoted"	5
"Beginning and ending with three single or double quote marks." "May span multiple lines and function like documents in shells"	
Feature "Anonymous functions"	5
automatic memory	1
yes indentation	-1

Showing 1 to 7 of 7 entries Previous 1 Next

The total weight associated is 59 .

Figure 17: Information extracted from Python features by LCE tool.

- **Complexity Considerations**

  The considerations, derived automatically by the tool (figure 17), are actually helpful to the users. It gives complexity assumptions, letting the end user to reason about the language's support.

  In terms of grammar, LCE tool produces the following complexity considerations:

  - The CFG is very large, in terms of productions and symbols, with many unitary productions and a low percentage of non-terminal recursive symbols, which contributes to its ease of understanding, manipulation, and maintenance.

  In terms of features, LCE tool produces the following complexity considerations:

  - In a language that does not require the use of semicolons, it makes the support process in a static analysis tool extremely complex, because it involves additional effort to figure out where an instruction begins and ends.
  - This language does not require explicit declaration of data type when creating a symbol. However, the language is strong-typed. This means that type inference needs to be done, making vulnerability detection more complex, as it is necessary to perform the compiler's work beforehand in order to know the information inferred by it, thus requiring more support effort.
  - This programming language has a scope policy defined by the code indentation, so the lexical analyser needs to do some work beforehand, since it needs to count the indentation, detect whether the block opens or closes, or whether it is the same, thus making the support process more time-consuming.

Considerations

About Grammar

Show entries Search:

Consideration
The CFG is very large, in terms of productions and symbols, with many unitary productions and a low percentage of non-terminal recursive symbols, which contributes to its ease of understanding, manipulation, and maintenance.

Showing 1 to 1 of 1 entries Previous 1 Next

About Knowledge Properties

Show entries Search:

Consideration
In a language that does not require the use of semicolons, it makes the support process in a static analysis tool extremely complex, because it involves additional effort to figure out where an instruction begins and ends.
This language by allowing a type declaration as dynamic does not require explicitly defining the data type when creating a symbol, making vulnerability detection more complex, as it is necessary to perform the compiler's work beforehand in order to know the information inferred by it, thus requiring more support effort.
This programming language requires its scopes to be delimited by their indentation, so the lexical analyzer needs to do some work beforehand, since it needs to count the indentation, detect whether the block opens or closes, or whether it is the same, thus making the support process more time consuming.

Showing 1 to 3 of 3 entries Previous 1 Next

Figure 18: Final Considerations about Python drawn by LCE tool.

Below, a new case study will be introduced and analysed, this time to compare the complexity of programming languages.

Example 12. *Description of the properties of the JavaScript programming language, written in PropLD DSL.*

```
1 JavaScript
2
3 PARADIGMS : event-driven, functional, imperative,
    procedural, object-oriented ;
4
5 DECLARATION : dynamic $7 ;
6
7 INDENTATION : no ;
8
9 SEMI-COLON : optional $9 ;
```

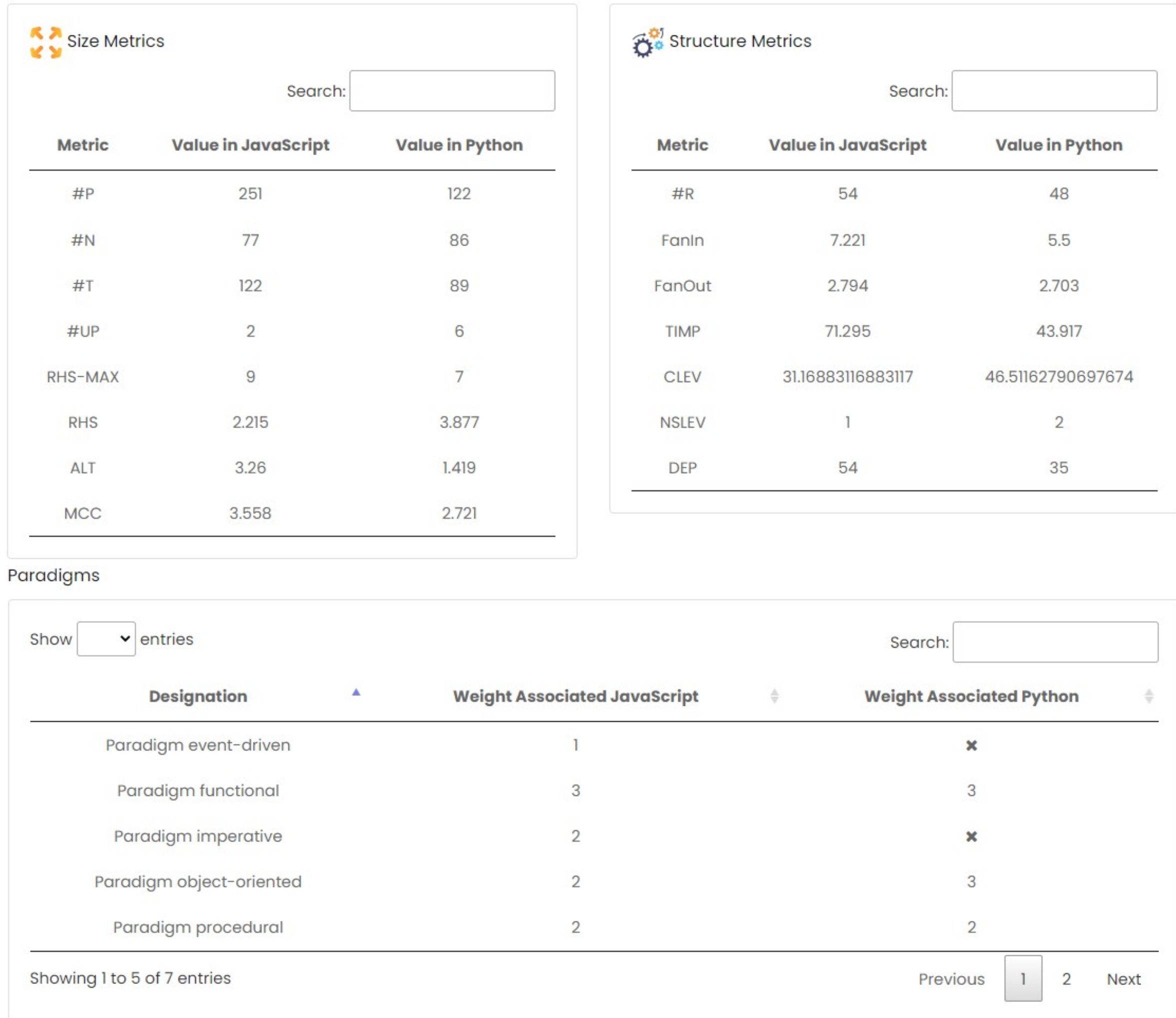
```
"Scoping" : [
    "Function scoping with 'var'",
    "Block scoping with the keywords 'let' and 'const'"
] $13;

"Weakly typed" : [
    "Means certain types are implicitly cast depending
        on the operation used"
] ;

"Anonymous function"  : [
    "A function definition that is not bound to an identifier.",
    "Anonymous functions are often arguments being passed to
        higher-order functions or used for constructing the result of
            a higher-order function that needs to return a function"
] ;
"Run-time environment" : [
    "typically relies on a run-time environment (e.g., a web browser)
        to provide objects and methods by which scripts can interact
            with the environment (e.g., a web page DOM)."
] ;
"First-class functions" : [
    "Function is considered to be an object",
    "A function may have properties and methods",
    "Created each time the outer function is invoked"
] ;

"Promises and Async/await" : [
    "Supports promises and Async/await for
        handling asynchronous operations."
]
.
```

Example 12 shows a description of the properties of another programming language, JavaScript, formatted according to the PropLD DSL presented in Chapter 3. Resorting to JavaScript grammar available in the same ANTLR grammars-v4 repository, already mentioned above, it is possible to make an automated and readable comparison between the two languages under consideration, using LCE tool.



Metric	Value in JavaScript	Value in Python
#P	251	122
#N	77	86
#T	122	89
#UP	2	6
RHS-MAX	9	7
RHS	2.215	3.877
ALT	3.26	1.419
MCC	3.558	2.721

Metric	Value in JavaScript	Value in Python
#R	54	48
FanIn	7.221	5.5
FanOut	2.794	2.703
TIMP	71.295	43.917
CLEV	31.1688311688311 7	46.51162790697674
NSLEV	1	2
DEP	54	35

Designation	Weight Associated JavaScript	Weight Associated Python
Paradigm event-driven	1	✖
Paradigm functional	3	3
Paradigm imperative	2	✖
Paradigm object-oriented	2	3
Paradigm procedural	2	2

Figure 19: Comparison of results between Python and JavaScript.

Figure 19 compares the metrics collected, and the various paradigms introduced, along with their associated weights. Aside from this information, feature descriptions with weights for each language may be found independently.

In summary, the grammar used to define JavaScript is far bigger than Python's one since it contains more productions and terminals; nevertheless, at the structural level, the JavaScript grammar is more like a graph in which the symbols are connected, whereas Python's is more like a tree. When it comes to attributes, the tool always depends on what the user describes, but based on these two samples, we can deduce that Python total weight was 49 and JavaScript total weight was 55.

6

CONCLUSION

This chapter is a summary and critical reflection of what has been presented so far. It also includes a discussion about some aspects of the research that are still needed, but that are relegated to future work.

The document begins by defining a SAST tool, being a software that analyses source code written in a programming language and identifies security vulnerabilities. It must be modified to handle various inputs when the source programming language changes in order to accomplish this. One of the main factors affecting the time to support a programming language is its complexity.

Then the motivation for this Master's work was presented. So, in order to assist language support teams in better estimating the time and effort required to support a new programming language using a parser, it was aimed to demonstrate that language complexity can be evaluated, using a tool to assess the difficulty of supporting a language based on a collection of metrics to classify the complexity of its grammar, as well as a set of features.

It was discussed that while estimating the effort required for a project is important, understanding and measuring software complexity is not something that is straightforward or obvious. However, it is possible to forecast the characteristics of the solution to a new problem, such as cost or time, by comparing the problems and taking into account the solutions discovered thus far.

For that, it was necessary to measure a software product's complexity in order to comprehend and control it for improvement; to do this, it is essential to classify the entities to be investigated, choose appropriate measurement objectives, and establish the maturity level attained.

Following this reasoning, it was determined that grammars would be a measurement of a language's complexity because they are the languages' creators and control how sentences in that language are recognized. Thus, when it comes to the complexity of the grammar, it was concluded that it concerns its size and how the symbols depend on each other; so a set of metrics was introduced to evaluate the complexity of context-free grammars.

The referred set of metrics can be divided into two subgroups: the Size and Structure metrics, both of which are considered useful for retaining information related to the complexity of the language being supported.

Based on the previous reasoning, the features of complexity for languages were determined because this thesis also aims to link a grammar's complexity to the complexity of the language it induces in the respective language.

It was necessary to conduct a study and analysis of the first processes that a SAST tool performs to achieve its objectives (detect vulnerabilities), in an initial phase in a generalized and then individualized way. In this case, the CxSAST tool from Checkmarx was examined.

Then, the language's complexity is determined by the complexity of its grammatical structure and semantic properties, which have an impact on the amount of time and effort required for support.

At this stage it became clear that, there are features of a programming language that grammatical metrics cannot assess or measure but which significantly affect support, particularly when it comes to maintain a static analysis tool.

In order to describe the features of a programming language that have an impact on its support, a DSL called *PropLD* was developed, to specify the core of linguistic features and paradigms.

This Master's work led to the development of a tool, **LCE - Language Complexity Evaluator**, available at `https://lce.di.uminho.pt`, that allows for the evaluation of a language's difficulty based on a set of metrics to rate the complexity of its grammar with a set of properties.

Furthermore, the tool analyses the difficulty of the new language as identified thus far with previously supported languages in order to forecast the effort required to process the new language.

Building the backend, required for applying the algorithms that allowed for the proper implementation of each module of the system's architecture, was the first priority. Then a frontend was put in place, which was in charge of extracting the values of each metric from other components and displaying them in a clear manner regarding the relevant language.

By examining the case study provided by LCE, it was determined that, in reality, it is possible to automatically judge the complexity of a grammar with using the metrics presented here and the DSL developed.

Keeping in mind the objectives outlined for this Master's project and reviewing the results, it is possible to deduce that from the identification of metrics that measure the syntactic complexity of a language and other language characteristics that impact on the processing of language sentences, it was implemented an evaluator to calculate the metrics identified in a repository to compare a new language with previously processed languages.

In this way it comes to the evidence that the project objectives were fully accomplished, and the hypothesis was proved.

Finally, it remains to mention that the research and the work developed made possible the writing of a scientific paper published and presented at the *11th Symposium on Languages, Applications and Technologies (SLATE 2022)* entitled "Determining Programming Languages Complexity and its impact on Processing" (Pinto et al., 2022).

This paper was written to show the basis of the LCE Tool, in other words to contextualize the reason for the creation of this tool, as well as the description and objectives of the project. Furthermore, it focuses on the main points to characterize the concepts that support this tool and presents a proposal that shows the architecture of the tool (at the time under development) and some results that could already be produced at the time of its publication.

6.1 FUTURE WORK

It is intended to discuss some details for a potential improvement to the developed tool, given the importance of the topic and the benefits its users want to extract from this tool to provide greater support in a SAST tool.

An important task, to be done in future work, is to improve the final considerations regarding grammatical complexity and description of the features provided. This may be accomplished by doing extensive testing on real programming languages and real users, similar to what was done in the case study described here, and comparing the outcomes with the conclusions of this dissertation.

BIBLIOGRAPHY

Tiago L. Alves and Joost Visser. Metrication of sdf grammars. Technical report, Universidade do Minho, 2005.

Tiago L. Alves and Joost Visser. A case study in grammar engineering. In *SLE*, 2008.

Roland Carl Backhouse. *Syntax of programming languages : theory and practice*. Englewood Cliffs : Prentice-Hall, 1979. ISBN: 0-13-879999-7.

Julien Cervelle, Matej Crepinsek, Rémi Forax, Tomaz Kosar, Marjan Mernik, and Gilles Roussel. On defining quality based grammar metrics. *2009 International Multiconference on Computer Science and Information Technology*, pages 651–658, 2009.

Checkmarx. Cxsast datasheet aug 2021. https://checkmarx.com/resources/data-sheet/cxsast-datasheet, August 2021a. Accessed on 13.12.2021.

Checkmarx. Why checkmarx - public sector 2021. https://checkmarx.com/resources/data-sheet/why-checkmarx-public-sector-2021, September 2021b. Accessed on 13.12.2021.

Noam Chomsky. Three models for the description of language. *IRE Trans. Inf. Theory*, 2: 113–124, 1956.

Kunal Chopra and Monika Sachdeva. Evaluation of software metrics for software projects. *International journal of computers & technology*, 14(6):5845–5853, 4 2015.

Don M. Coleman, Dan Ash, Bruce Lowther, and Paul W. Oman. Using metrics to evaluate software system maintainability. *Computer*, 27:44–49, 1994.

Matej Crepinsek, Tomaz Kosar, Marjan Mernik, Julien Cervelle, Rémi Forax, and Gilles Roussel. On automata and language based grammar metrics. *Comput. Sci. Inf. Syst.*, 7: 309–329, 2010.

João Cruz. Qge - an attribute grammar based system to assess grammars quality. Master's thesis, Universidade do Minho, 12 2015.

Erzsébet Csuhaj-Varjú and Alica Kelemenová. Descriptional complexity of context-free grammar forms. *Theor. Comput. Sci.*, 112:277–289, 1993.

Norman E. Fenton and Martin Neil. Software metrics: roadmap. In *ICSE '00*, 2000.

Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics : A Rigorous and Practical Approach*. Boston : PWS Publishing Company, 2 edition, 1997. ISBN: 0-534-95425-1.

R. Finkbine. Metrics and models in software quality engineering. *ACM Sigsoft Software Engineering Notes*, 21:89, 1996.

Pedro Rangel Henriques. Brincando às linguagens com rigor: Engenharia gramatical. Technical report, Departamento de Informática da Escola de Engenharia da Universidade do Minho, 11 2013.

ISO/IEC. Iso/iec/ieee international standard for software engineering - software life cycle processes - maintenance. *ISO/IEC 14764:2006 (E) IEEE STD 14764-2006 Revision of IEEE STD 1219-1998)*, pages 1–58, 2006. doi: 10.1109/IEEESTD.2006.235774.

Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison Wesley, 2 edition, 2002. ISBN: 0-201-72915-6.

Donald Ervin Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2: 127–145, 2005.

Jinfeng Li. Vulnerabilities mapping based on owasp-sans: a survey for static application security testing (sast). *ArXiv*, abs/2004.03216, 2020.

Sofia Nystedt and Claes Sandros. Software complexity and project performance. Master's thesis, University of Gothenburg, 1999.

Terence John Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. The Pragmatic Bookshelf, 2009. ISBN: 978-1-934356-45-6.

Terence John Parr. *The Definitive ANTLR 4 Reference*. The Pragmatic Bookshelf, 2013. ISBN: 978-1-93435-699-9.

Gonçalo Pinto, Pedro Rangel Henriques, Daniela da Cruz, and João Cruz. Determining Programming Languages Complexity and its impact on Processing. In João Cordeiro, Maria João Pereira, Nuno Rodrigues, and Sebastiao Pais, editors, *11th Symposium on Languages, Applications and Technologies (SLATE 2022)*, volume 104 of *Open Access Series in Informatics (OASIcs)*, pages 16:1–16:15, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-245-7. doi: 10.4230/OASIcs.SLATE.2022.16. URL https://drops.dagstuhl.de/opus/volltexte/2022/16762.

J.F. Power and Brian A. Malloy. Metric-based analysis of context-free grammars. *Proceedings IWPC 2000. 8th International Workshop on Program Comprehension*, pages 171–178, 2000.

J.F. Power and Brian A. Malloy. A metrics suite for grammar-based software. *J. Softw. Maintenance Res. Pract.*, 16:405–426, 2004.

Stephen R. Schach. *Object-Oriented and Classical Software Engineering*. McGraw-Hill, Inc., 8 edition, 2010. ISBN 978-0-13-394302-3.

John W. Tukey. The teaching of concrete mathematics. *American Mathematical Monthly*, 65: 1–9, 1958.

ANOTHER TOOL FOR LANGUAGE RECOGNITION (ANTLR)

The goal of this appendix is to provide a brief overview of ANTLR v4 Tool, developed by Terrence Parr and Sam Harwell, so that the reader can quickly become acquainted with hat Compilers Generator system, what it does and how it does it, and thus answer some questions that may arise in the reader's mind while reading this dissertation.

To recognize a language, a software that reads sentences and responds appropriately to input words and symbols must be built. In general, an interpreter is a program that calculates or "executes" phrases. If the procedure involves translating sentences from one language to another, the application is known as a translator.

To respond effectively, an interpreter or translator must recognize all legitimate sentences and subordinate clauses in a particular language. Identifying a sentence implies being able to recognize its many components and distinguish it from other sentences.

Parsers and syntax analysers are programs that recognize languages. The principles that control linguistic association are referred to as syntax. A grammar is just a set of rules that convey the structure of a sentence in different ways. ANTLR is a program that converts grammars to parsers; in other words, ANTLR is a program that produces other programs. The syntax of the ANTLR meta-language must be followed by this input grammar.

Parsing becomes significantly easier when we divide it into two comparable but different phases.

- The first phase, known as lexical analysis, relates to the process of grouping letters into words or symbols (tokens), which includes the token type (identifying the lexical structure) and the text matching to that token by the Lexer, which is a software that tokenizes the input.
- The second phase is the real Parser, which uses the previous phase's tokens to detect the sentence structure. To generate these parsers, the ANTLR tool creates a data structure called a parse tree or syntax tree, which explains how the parser detected the input phrase and its components.

A.1 COMPONENTS

As explained previously, the ANTLR tool creates the code for the user. This operation is broken into two parts, each of which involves the use of a distinct ANTLR component.

- The first part is concerned with the process of running the ANTLR tool, which builds Parser and Lexer code that identifies phrases in the language represented by the given grammar.
- In the second part, the resulting files are built and executed using the ANTLR runtime API, which is a library of classes and functions.

ANTLR creates a variety of files from the grammar (.g4 extension file), including:

- *(NameOfGrammar)Parser* provides the specification of a grammar-specific parser class that recognizes the syntax of the language and specifies a method for each grammar rule, as well as some supporting code;
- *(NameOfGrammar)Lexer*, which clearly includes the Lexer class definition;
- *(NameOfGrammar).tokens*, for each token that has been defined, the token type number is recorded in this file;
- The interface *(NameOfGrammar)Listener* defines the callbacks[1] that can be implemented;
- *(NameOfGrammar)BaseListener* is a collection of empty methods that the user must fill up in order to generate the implementations;

Figure 20: ANTLR creates a variety of files from the grammar.

If these files can be used in a primary program built in Java, this sort of tool becomes extremely valuable in a language application, such as the one described in this work. ANTLR has methods for this, and it was these methods that were utilized to produce / execute the Parser generated by the supplied grammar. Consider the following example, which was utilized in the LCE application during the stage of reading the specified files.

Example 13. *ANTLR tool integration in a Java main application.*

1 ANTLR parser generate a tree from the input by default. A tree walker can send "events" (callbacks) to a listener object by walking through it.

```java
import org.antlr.v4.runtime.CharStreams;
import org.antlr.v4.runtime.CommonTokenStream;
import java.io.IOException;

public class Main {
  public static void main(String[] args){
    try {
      XXXLexer lexer = new XXXLexer(CharStreams.fromFileName("PathInputFile.txt"));
      CommonTokenStream stream = new CommonTokenStream(lexer);
      XXXParser parser = new XXXParser(stream);
      parser.AXIOM-OF-XXX();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Listing A.1: XXX Language Processor

A.2 GRAMMAR

This section will offer a brief overview of the ANTLR meta language. Although not every feature of the language will be discussed, the material provided should be sufficient to comprehend the grammar of language and how to develop one.

A.2.1 *Lexicon*

Most programmers are familiar with ANTLR's lexicon, since it follows the syntax of C and its derivatives with some additions for grammatical descriptions.

- **Comments** - there are three types of comments: single-line, multiline, and Javadoc-style. The Javadoc comments are not ignored and are delivered to the parser. These are only permitted at the beginning of a grammatical rule or at the beginning of any rule;
- **Identifiers** - lexer rules, like token names, always begin with a capital letter. The name of a parser rule must always begin with a lower case letter. Upper and lower case letters, numbers, and underscores can be used after the first character;
- **Literals** - ANTLR, unlike other languages, does not differentiate between character and string literals. Literals may include Unicode escape sequences of the type uXXXX, where XXXX is the hexadecimal Unicode character value.

Unicode's characters can be used directly inside literals or via the use of Unicode escape sequences. ANTLR develops recognizers based on a character vocabulary that includes all Unicode characters;

- **Keywords** - the following terms are reserved in ANTLR grammars: *import, fragment, lexer, parser, grammar, returns, locals, throws, catch, finally, mode, options, tokens*. Furthermore, even if it is not a keyword, the term rule should not be used as an alternative rule or label name. Additionally, no target language terms should be utilized as a symbol, label, or rule name.

A.3 Structure

A grammar is essentially a grammar declaration followed by a list of rules. The file containing the X grammar should be called X.g4. The sequence of options, imports, token specifications, and actions is not important. Each option, import, and token specification can only have one. Except for the header and at least one rule, all of these parts are optional.

The names of parser rules must begin with a lowercase letter, whereas lexer rules must begin with a capital letter. A combined grammar can include lexical and parser rules, for this purpose grammars are defined without a prefix in their header.

A.4 Parser rules

In ANTLR, parser rules are relatively simple to read, write, and comprehend. With a few minor exceptions, rules are authored in the same manner as the production. When there are several productions for the same non-terminal, they must all be expressed in a single parser rule, with alternatives utilized from the second to the final production of that non-terminal.

All rules in ANTLR must finish with ';', and there are several techniques available, such as naming the alternatives, identifying the constituents of a rule, and introducing subrules. Tokens (starting with an uppercase letter), string literals, and non-terminals for other rules can exist on the right side of a rule (starts with a lowercase letter)

A.5 Lexer rules

Lexer rules, like parser rules, define token definitions and have a comparable syntax. ANTLR offers a variety of ways to describe lexical rules, including lexical modes, recursive rules, and lexer commands. On the right-hand side of lexer rules, a literal that corresponds to a sequence of characters, a character set that corresponds to one of the characters in that set, or activate a lexer rule, among other things, can be used.

B

PROPLD LEXICAL RULES

In Section 3.1 a DSL to describe the properties of a programming language that can impact on its processing, called PropLD, was introduced. Its syntax was formally specified via a set of grammar rules written according to ANTLR metalanguage.

This appendix presents in Listing B.1 the lexer rules used to specify the Terminal symbols referred to in those grammar syntactic rules. The rules below allow for the automatic generation of the Lexer Analyser necessary to process PropLD descriptions. The generation process was implemented, resorting to ANTLR tool.

Lexer rules use Regular Expressions over the character set to define the way the language Tokens (Terminal symbols) are written, this is, lexer rules define the language orthography. According to ANTLR meta-grammar, lexer rule names must begin with an uppercase letter to be distinguished from parser rule names. It is feasible to construct rules that are not tokens, but rather help in token recognition. This purpose is attained using fragment rules that produce no tokens accessible to the parser.

```
DOT: '.';

RPAR: ']';

COMMA: ',';

LPAR: '[';

COLON: ':';

SEMICOLON: ';';

DOLLAR: '$';

OPTIONAL: 'Optional' | 'optional' | 'OPTIONAL';

STRSEMICOLON: 'Semi-colon' | 'semi-colon' | 'SEMI-COLON';

MANUAL: 'Manual' | 'manual' | 'MANUAL';
```

```
MEMORY: 'Memory' | 'memory' | 'MEMORY';

NO: 'No' | 'no' | 'NO';
YES: 'Yes' | 'yes' | 'YES';

INDENTATION: 'Indentation' | 'indentation' | 'INDENTATION';

BOTH: 'Both' | 'both' | 'BOTH';
DYNAMIC: 'Dynamic' | 'dynamic' | 'DYNAMIC';
STATIC: 'Static' | 'static' | 'STATIC';

DECLARATION: 'Declaration' | 'declaration' | 'DECLARATION';

OTHER: 'Other' | 'other' | 'OTHER';
ACTION: 'Action' | 'action' | 'ACTION';
AGENTORIENTED: 'Agent-oriented' | 'agent-oriented' | 'AGENT-ORIENTED';
ARRAYORIENTED: 'Array-oriented' | 'array-oriented' | 'ARRAY-ORIENTED';
AUTOMATABASED: 'Automata-based' | 'automata-based' | 'AUTOMATA-BASED';
CONCURRENTCOMPUTING: 'Concurrent Computing' | 'concurrent computing' | '
    CONCURRENT COMPUTING';
CHOREOGRAPHICPROGRAMMING: 'Choreographic Programming' | 'choreographic
    programming' | 'CHOREOGRAPHIC PROGRAMMING';
RELATIVISTICPROGRAMMING: 'Relativistic Programming' | 'relativistic programming'
    | 'RELATIVISTIC PROGRAMMING';
DATADRIVEN: 'Data-driven' | 'data-driven' | 'DATA-DRIVEN';
DECLARATIVE: 'Declarative' | 'declarative' | 'DECLARATIVE';
FUNCTIONAL: 'Functional' | 'functional' | 'FUNCTIONAL';
FUNCTIONALLOGIC: 'Functional Logic' | 'functional logic' | 'FUNCTIONAL LOGIC';
PURELYFUNCTIONAL: 'Purely Functional' | 'purely functional' | 'PURELY FUNCTIONAL'
    ;
LOGIC: 'Logic' | 'logic' | 'LOGIC';
ABDUCTIVELOGIC: 'Abductive Logic' | 'abductive logic' | 'ABDUCTIVE LOGIC';
ANSWERSET: 'Answer Set' | 'answer set' | 'ANSWER SET';
CONCURRENTLOGIC: 'Concurrent Logic' | 'concurrent logic' | 'CONCURRENT LOGIC';
INDUCTIVELOGIC: 'Inductive Logic' | 'inductive logic' | 'INDUCTIVE LOGIC';
CONSTRAINT: 'Constraint' | 'constraint' | 'CONSTRAINT';
CONSTRAINTLOGIC: 'Constraint Logic' | 'constraint logic' | 'CONSTRAINT LOGIC';
CONCURRENTCONSTRAINTLOGIC: 'Concurrent Constraint Logic' | 'concurrent constraint
     logic' | 'CONCURRENT CONSTRAINT LOGIC';
DATAFLOW: 'Dataflow' | 'dataflow' | 'DATAFLOW';
FLOWBASED: 'Flow-based' | 'flow-based' | 'FLOW-BASED';
REACTIVE: 'Reactive' | 'reactive' | 'REACTIVE';
FUNCTIONALREACTIVE: 'Functional Reactive' | 'functional reactive' | 'FUNCTIONAL
    REACTIVE';
ONTOLOGY: 'Ontology' | 'ontology' | 'ONTOLOGY';
QUERYLANGUAGE: 'Query Language' | 'query language' | 'QUERY LANGUAGE';
```

```
DIFFERENTIABLE: 'Differentiable' | 'differentiable' | 'DIFFERENTIABLE';
DYNAMICSCRIPTING: 'Dynamic/scripting' | 'dynamic/scripting' | 'DYNAMIC/SCRIPTING'
    ;
EVENTDRIVEN: 'Event-driven' | 'event-driven' | 'EVENT-DRIVEN';
FUNCTIONLEVEL: 'Function-level' | 'function-level' | 'FUNCTION-LEVEL';
POINTFREESTYLE: 'Point-free Style' | 'point-free style' | 'POINT-FREE STYLE';
CONCATENATIVE: 'Concatenative' | 'concatenative' | 'CONCATENATIVE';
GENERIC: 'Generic' | 'generic' | 'GENERIC';
IMPERATIVE: 'Imperative' | 'imperative' | 'IMPERATIVE';
PROCEDURAL: 'Procedural' | 'procedural' | 'PROCEDURAL';
OBJECTORIENTED: 'Object-oriented' | 'object-oriented' | 'OBJECT-ORIENTED';
POLYMORPHIC: 'Polymorphic' | 'polymorphic' | 'POLYMORPHIC';
INTENTIONAL: 'Intentional' | 'intentional' | 'INTENTIONAL';
LANGUAGEORIENTED: 'Language-oriented' | 'language-oriented' | 'LANGUAGE-ORIENTED'
    ;
DOMAINSPECIFIC: 'Domain-specific' | 'domain-specific' | 'DOMAIN-SPECIFIC';
LITERATE: 'Literate' | 'literate' | 'LITERATE';
NATURALLANGUAGEPROGRAMMING: 'Natural-language Programming' | 'natural-language
    programming' | 'NATURAL-LANGUAGE PROGRAMMING';
METAPROGRAMMING: 'Metaprogramming' | 'metaprogramming' | 'METAPROGRAMMING';
AUTOMATIC: 'Automatic' | 'automatic' | 'AUTOMATIC';
INDUCTIVEPROGRAMMING: 'Inductive Programming' | 'inductive programming' | '
    INDUCTIVE PROGRAMMING';
REFLECTIVE: 'Reflective' | 'reflective' | 'REFLECTIVE';
ATTRIBUTEORIENTED: 'Attribute-oriented' | 'attribute-oriented' | 'ATTRIBUTE-
    ORIENTED';
MACRO: 'Macro' | 'macro' | 'MACRO';
TEMPLATE: 'Template' | 'template' | 'TEMPLATE';
NONSTRUCTURED: 'Non-structured' | 'non-structured' | 'NON-STRUCTURED';
ARRAY: 'Array' | 'array' | 'ARRAY';
NONDETERMINISTIC: 'Nondeterministic' | 'nondeterministic' | 'NONDETERMINISTIC';
PARALLELCOMPUTING: 'Parallel Computing' | 'parallel computing' | 'PARALLEL
    COMPUTING';
PROCESSORIENTED: 'Process-oriented' | 'process-oriented' | 'PROCESS-ORIENTED';
PROBABILISTIC: 'Probabilistic' | 'probabilistic' | 'PROBABILISTIC';
QUANTUM: 'Quantum' | 'quantum' | 'QUANTUM';
SETTHEORETIC: 'Set-theoretic' | 'set-theoretic' | 'SET-THEORETIC';
STACKBASED: 'Stack-based' | 'stack-based' | 'STACK-BASED';
STRUCTURED: 'Structured' | 'structured' | 'STRUCTURED';
BLOCKSTRUCTURED: 'Block-structured' | 'block-structured' | 'BLOCK-STRUCTURED';
STRUCTUREDCONCURRENCY: 'Structured Concurrency' | 'structured concurrency' | '
    STRUCTURED CONCURRENCY';
ACTORBASED: 'Actor-based' | 'actor-based' | 'ACTOR-BASED';
CLASSBASED: 'Class-based' | 'class-based' | 'CLASS-BASED';
CONCURRENT: 'Concurrent' | 'concurrent' | 'CONCURRENT';
PROTOTYPEBASED: 'Prototype-based' | 'prototype-based' | 'PROTOTYPE-BASED';
```

```
100 BYSEPARATIONOFCONCERNS: 'By Separation Of Concerns' | 'by separation of concerns'
    | 'BY SEPARATION OF CONCERNS';
ASPECTORIENTED: 'Aspect-oriented' | 'aspect-oriented' | 'ASPECT-ORIENTED';
102 ROLEORIENTED: 'Role-oriented' | 'role-oriented' | 'ROLE-ORIENTED';
SUBJECTORIENTED: 'Subject-oriented' | 'subject-oriented' | 'SUBJECT-ORIENTED';
104 RECURSIVE: 'Recursive' | 'recursive' | 'RECURSIVE';
SYMBOLIC: 'Symbolic' | 'symbolic' | 'SYMBOLIC';
106 VALUELEVEL: 'Value-level' | 'value-level' | 'VALUE-LEVEL';

108 PARADIGM: 'Paradigm' | 'paradigm' | 'PARADIGM' | 'Paradigms' | 'paradigms' | '
    PARADIGMS';

110 NUM: '-'? [0-9]+;

112 STR: '"' ( ~('"'))* '"';

114 VERSION: [0-9]+ (('.' [0-9]+)+)?;

116 ID: [a-zA-Z]+;

118 WS: ('\r'? '\n' | ' ' | '\t')+ -> skip;
```

Listing B.1: Lexer Rules of the PropLD.